VISUAL TIME-SAVING REFERENCE

Core Data Quick Start

iOS 16

In SwiftUI

Mark Moeykens

A CORE DATA REFERENCE GUIDE FOR SWIFTUI DEVELOPERS

Big Mountain Studio

Version: 22-JUNE-2022

ACKNOWLEDGEMENTS

Writing a book while also working a full-time job takes time away from family and friends. So first of all, I would like to thank my wife Jaqueline and daughter Paola for their patience and support.

Next, I would like thank Chris Ching who finally convinced me to learn Core Data and instructed me on this topic.

I would also like to thank my **friends** who always gave me constant feedback, support, and business guidance: Chris Ching, Scott Smith, Rod Liberal, Chase Blumenthal and Chris Durtschi.

I would also like to thank the **Utah developer community** for their help in making this book possible. This includes Dave DeLong and Andrew Madsen.

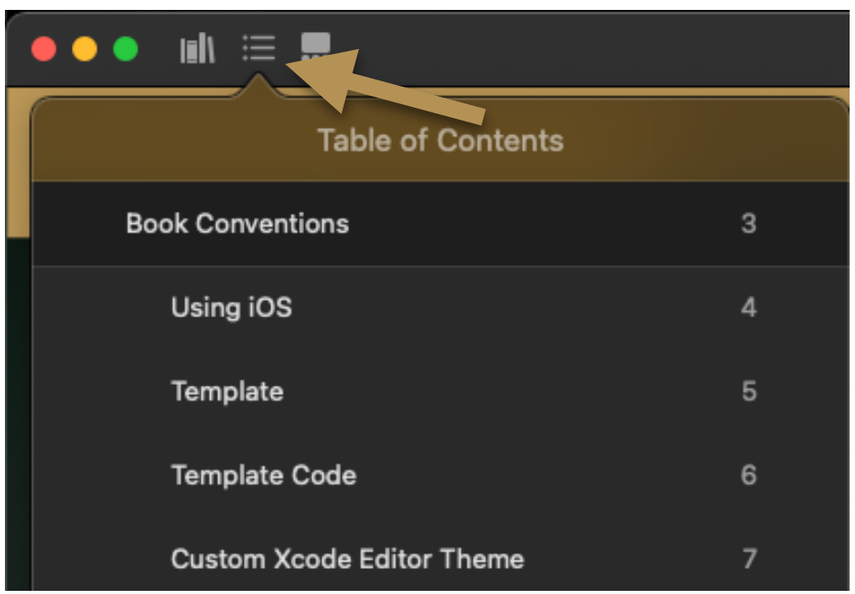
Many other **developers** also proof-read and gave feedback on the book. These include: Stewart Lynch, Waleed Alware, Antti Juustila, Chris Gambrell, Jahns Hendrik, Terrance McHugh, Gerard Gomez, Berkant Dursan, Darius Dunlap, Robert DeLaurentis, Jake Steramma, and Inal Gotov.

And finally, I would like to thank the **creators** of all the other sources of information, whether Swift or Core Data, that really helped me out and enabled me to write this book. That includes Apple and their documentation and definition files, Donny Wals, Antoine van der Lee, Paul Hudson, and Mohammad Azam.

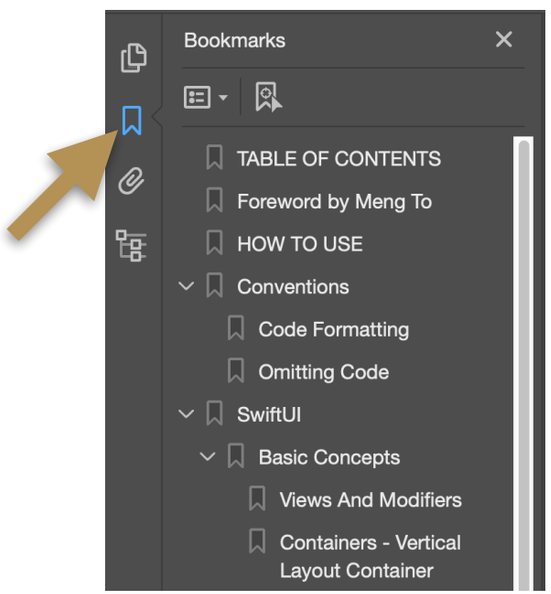
Table of Contents

The table of contents should be built into your EPUB and PDF readers. Examples:

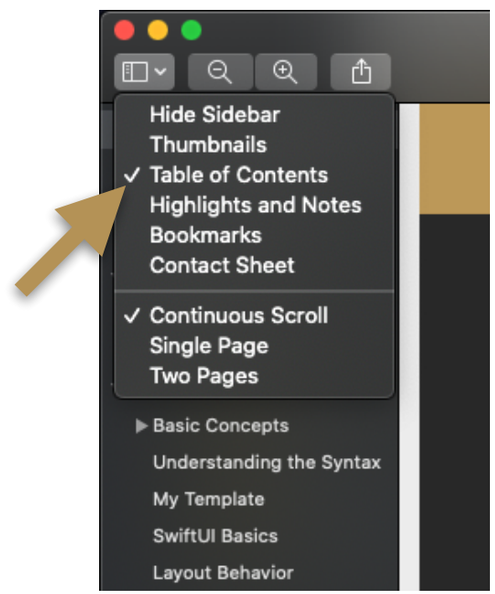
Books App



Adobe Acrobat Reader



Preview

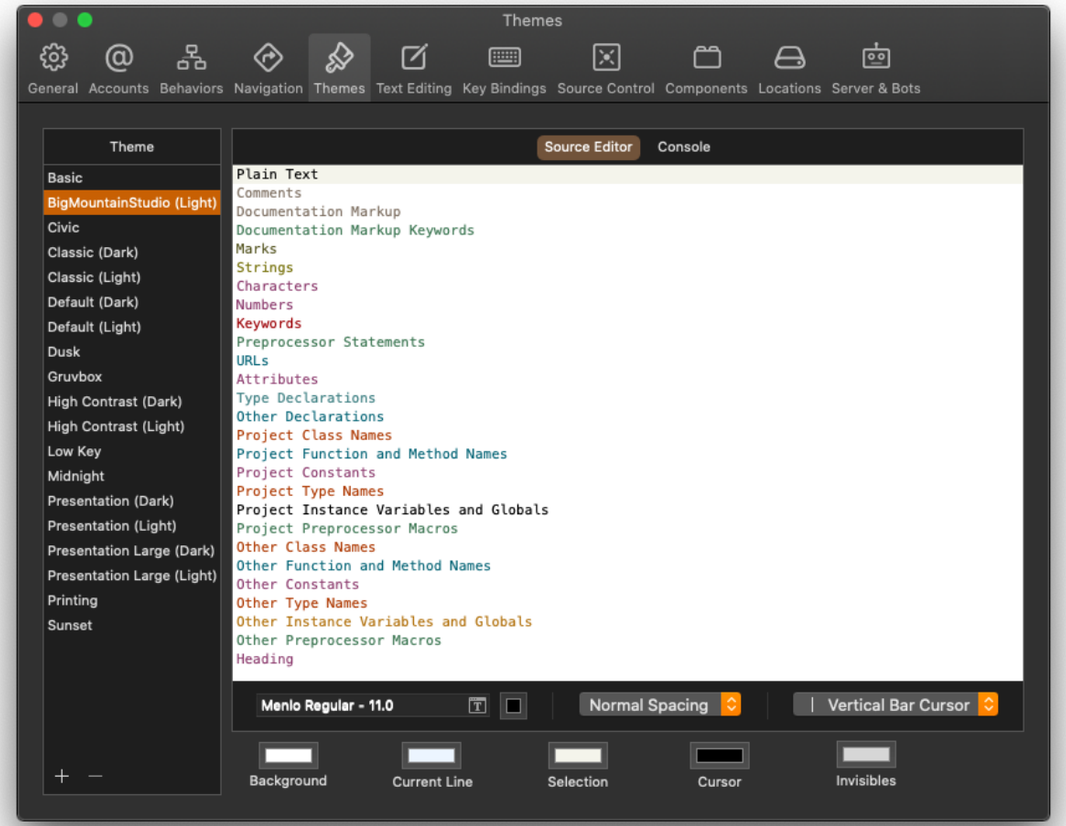


BOOK CONVENTIONS

Custom Xcode Editor Theme

I created a code editor color theme for a high-contrast light mode. This is the theme I use for the code throughout this book.

If you like this color theme and would like to use it in your Xcode then you can find it on my GitHub as a gist here.



If you download the theme from the gist, look at the **first line** (comment) for where to put it so Xcode can see it.

If the directory does not exist, then you will have to create it.

Embedded Videos

The **ePUB** version of the book supports embedded videos.

The **PDF** version does not.

This icon indicates that this is a playable video in the ePUB format.

But in PDF it renders as simply a screenshot.

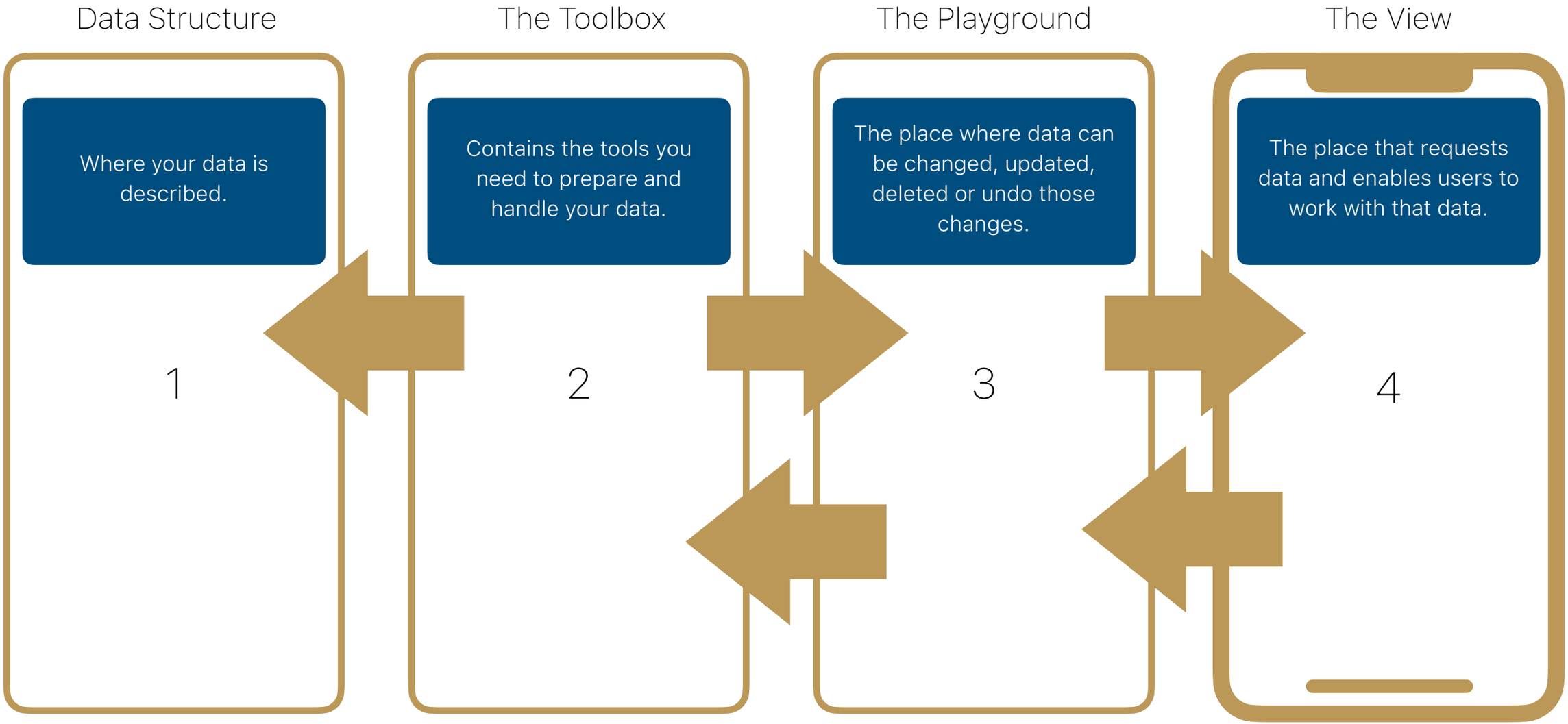
In some ePUB readers, including Apple Books, you might have to **tap TWICE** (2) to play the video.

CORE DATA CONCEPTS

There are similar patterns that can be found when you have applications working with data.
Core Data is no different.
First, you will learn some concepts. No code in this chapter.
Then you will learn what Apple calls each of these parts in the Core Data framework. In the end, you will be able to THINK with Core Data concepts.

The Flow

Let's go over the 4 main components used to work with Core Data in SwiftUI. You will learn the **real names** and purposes behind these concepts. We won't get too technical. This chapter will give you a good overview of the pieces that make up Core Data and how to think with it.

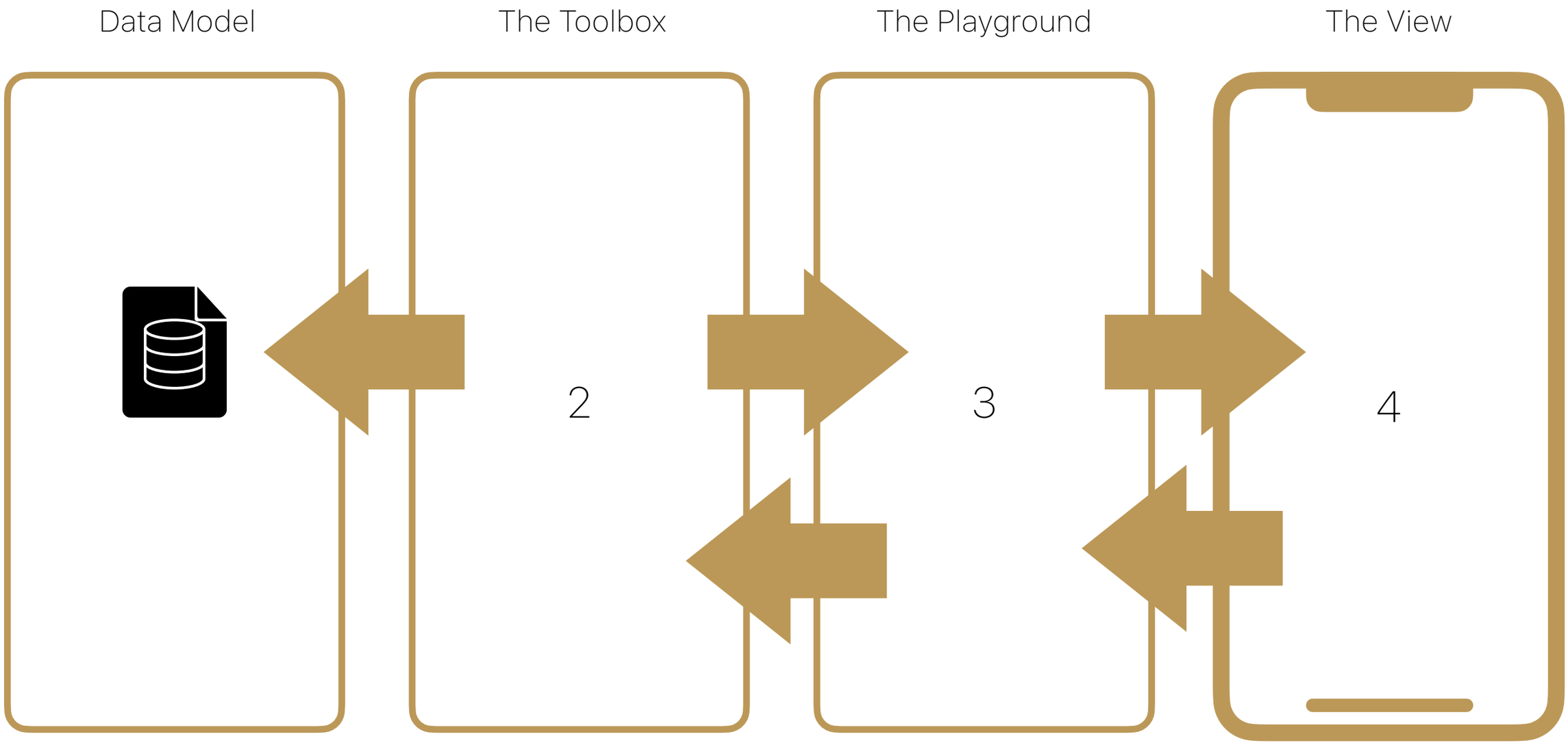


1. Data Model

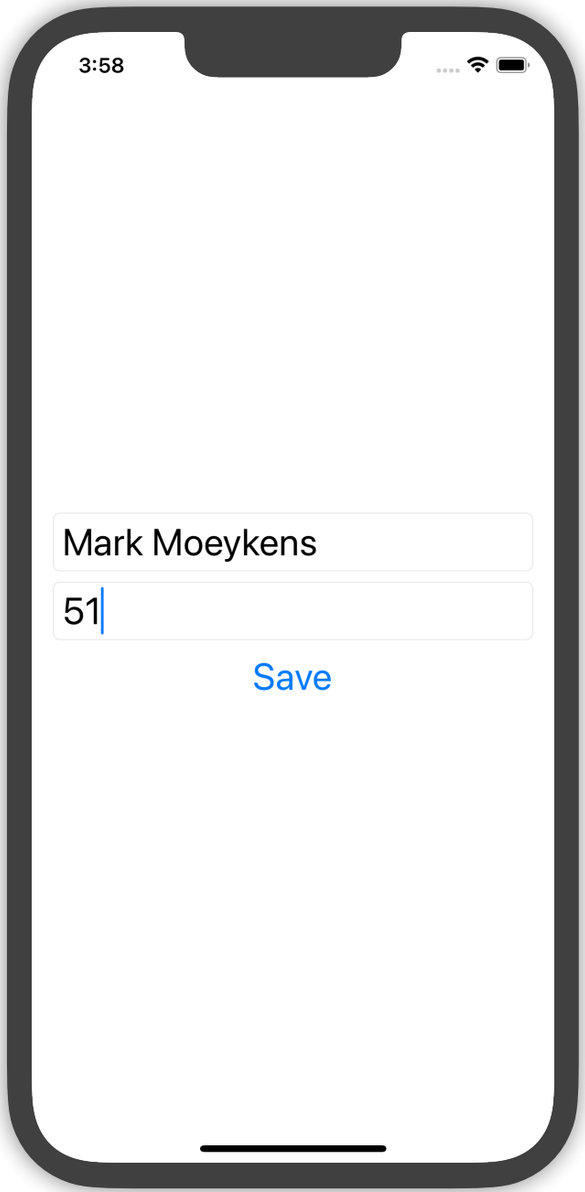
The first part of the flow is the data model. You will learn what a data model is (and what it is not).

Data Models - The First Part

The first part is what is called the “Data Model”. A data model tells Core Data what the data looks like.



Data Models




```
struct Person {
    let name: String
    let age: Int
}
```

This defines a **data model** but there is no actual data in it.

```
struct DataModels: View {
    @State private var name = ""
    @State private var age = 0

    var body: some View {
        VStack {
            TextField("Name", text: $name)
            TextField("Age", value: $age, format: .number)

            Button("Save") {
                let newPerson = Person(name: name, age: age)

                // Save data object somewhere
            }
        }
        .textFieldStyle(.roundedBorder)
        .padding()
        .font(.title)
    }
}
```

You have probably created a struct or a class to hold data.

These classes and structs are called **data models**.

Data models are populated with values and used in some way.

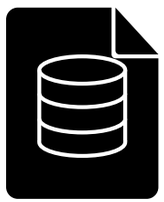
Core Data needs to know how to store a `Person` object.

You need to define your data models in a special way before working with them in Core Data.

Data Models or Data Objects?

Let's clear up the difference between "data model" and "data object".

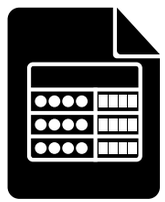
Data Model



The **model** is the definition of how data will be structured or organized. It defines property names and types. It can be a struct or a class.

```
struct Person {
    let name: String
    let age: Int
}
```

Data Object



The **object** is an instance of the model that is populated with data. For example, the `newPerson` variable here is an object:

```
let newPerson = Person(name: name, age: age)
```

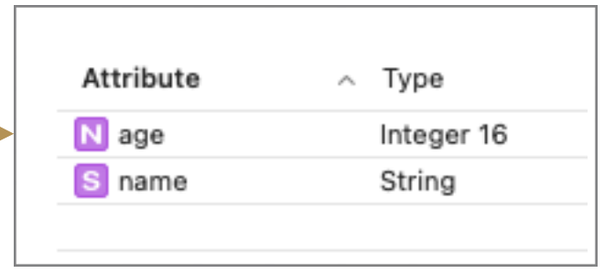
Core Data can't use the Person struct as it is defined here. You will have to define it a different way. **See next page**.

Define Your Data Models

You define your data model descriptions in a special “Data Model” file that Core Data uses.

This helps Core Data understand how you want your data structured.

This book will walk you through how to use this and set it up.



The data model

This file just holds the definition of objects, like property names and their types (string, bool, etc).

It doesn’t actually hold any values.

This will actually replace the Person struct on the previous page.

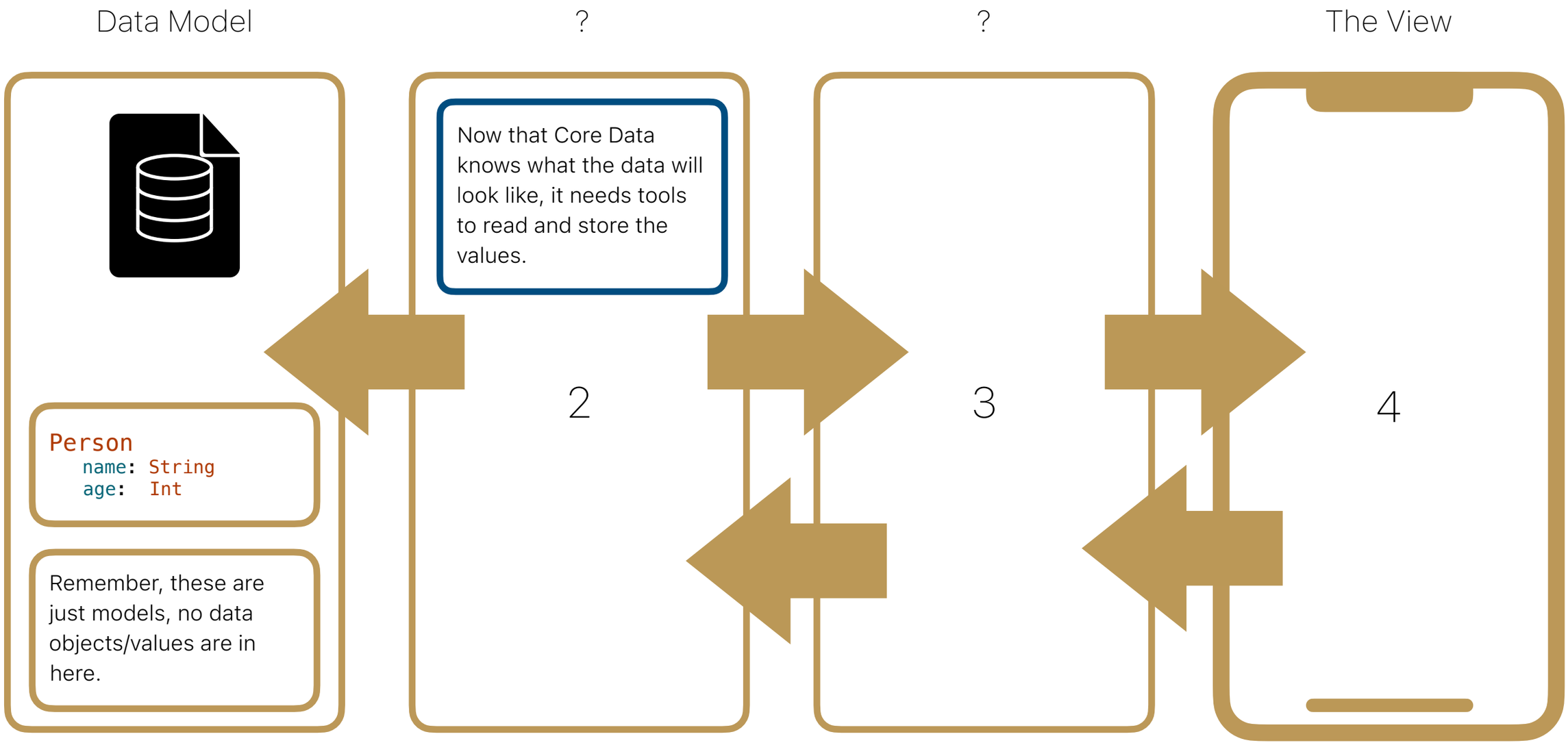
Think of “**Attribute**” as “property”.

The purple icons just represent the data types specified.

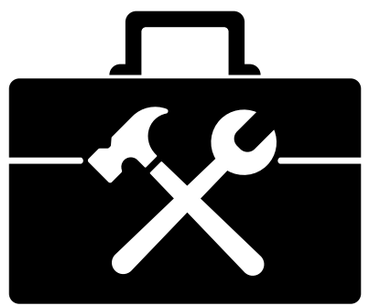
N = Number
S = String

Part 1 in The Flow: Data Model

So the first thing you need is a data model so Core Data knows the structure of your data, properties, types, etc.



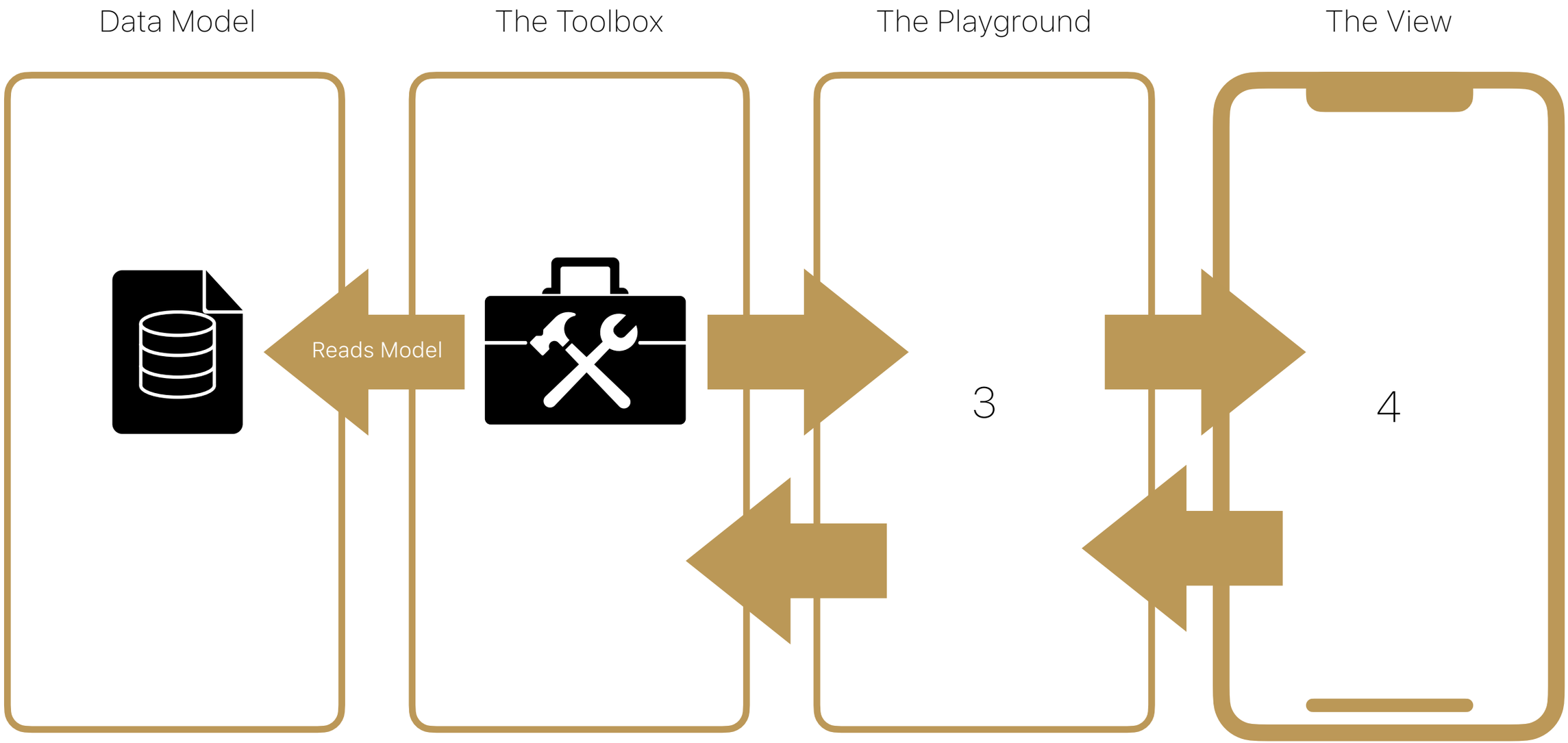
2. Tools for Persistence



When your app has data, you have to store it someplace where it can persist so when the app is closed or if the device loses power it won't disappear. This chapter talks about the subject of persistence.

The Toolbox - The Second Part

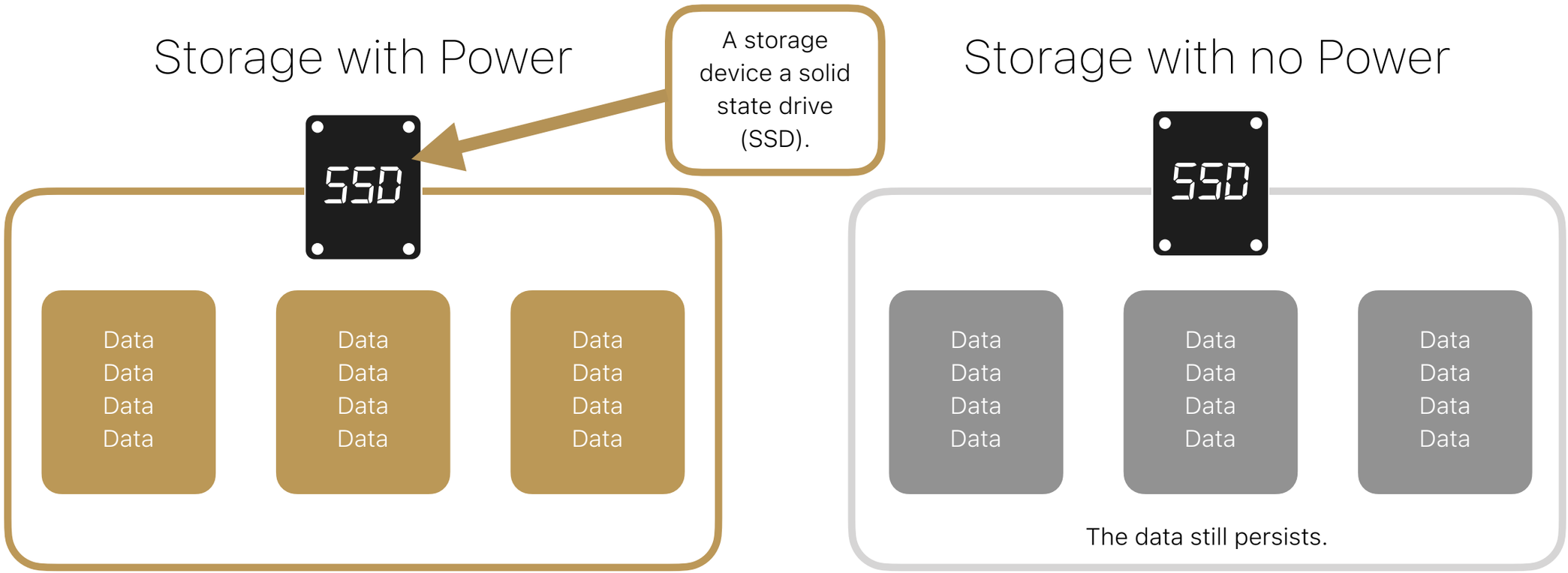
"Persist" means "to continue to survive or exist". If you close your app or turn your phone off, the data will survive.
The second part, the toolbox, has to do with tools to read your data model and with this knowledge it can retrieve and persist your data.



Persistent Storage

When an app saves data it will store it somewhere so that data can then persist, even when the power is turned off or the app is closed.

Apps on your iPhone are stored so they can persist if your phone loses all power. All iPhones have a physical component that can store apps and data.



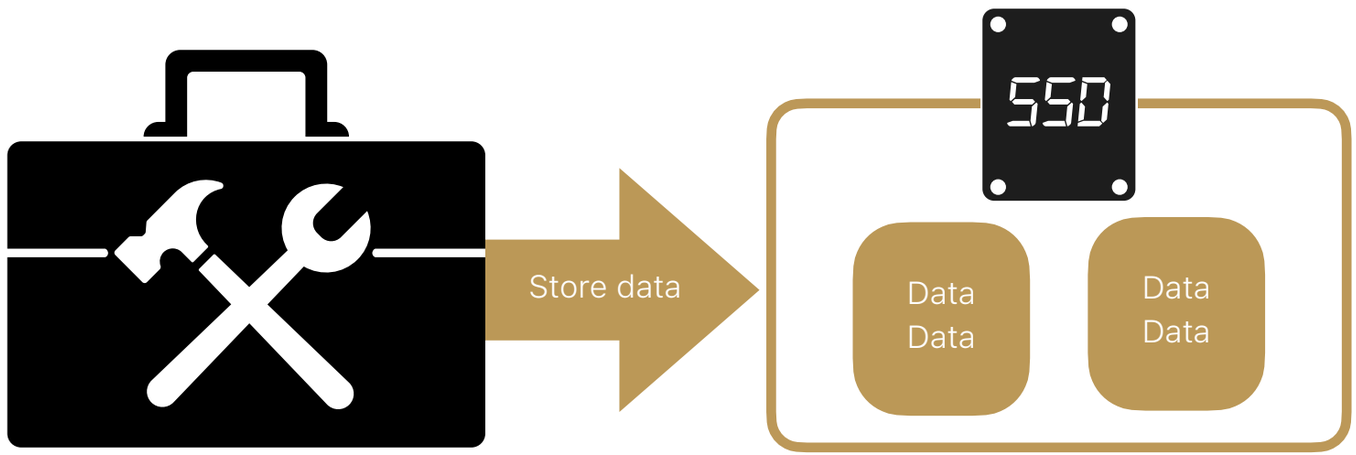
Core Data will store values to the physical storage on the device so it can persist after closing the app or shutting off the device.

NSPersistentContainer

Core Data uses the **NSPersistentContainer** to store data so it persists on the device. **This IS the toolbox.** The NSPersistentContainer has quite a few tools we can use.

NSPersistentContainer

Persistent Storage



The NSPersistentContainer:

- Reads your Data Model and sets it up for use
- Controls where your data is persisted on the device
- Is responsible for loading up your data from the persistent storage so your app can use it

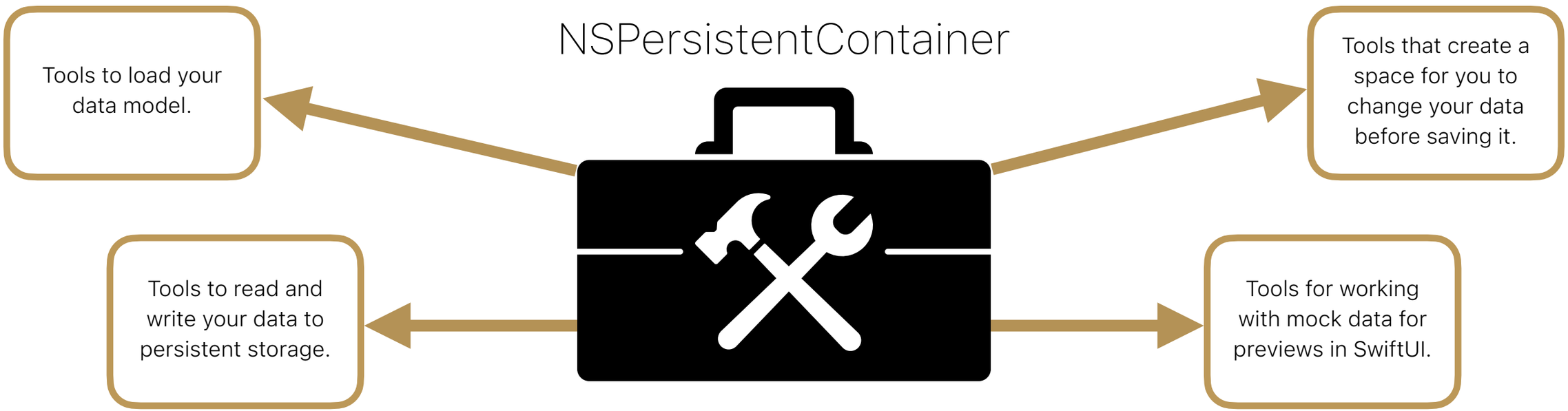
Note: NS stands for "**N**ext **S**tep" (NeXTSTEP) which was an operating system that Apple bought in 1996 and became the foundation of macOS in 2000. Yeah, it's old. But it's continually being worked on.

Container

Think of a **container** as a box that can hold many things. A container can hold tools, food, pencils, etc.

In programming, containers can hold a collection of other objects.

The NSPersistent**Container** is like a toolbox (container) that will provide you with almost every tool you might need to work with Core Data.



Many of these tools were separate. Developers had to do a lot of setup work to make them all work together in the past.

But Apple developers made Core Data way easier to use with the NSPersistentContainer.

Data Model + Data Values = Data Objects

Your data model and the actual data values are two separate files. The persistent container and its toolbox will “merge” the two together. It always gives you the data, as defined in your data model, so you have data objects you can now display and work with.

Data Model

Describes what the data will look like.

Person	
name	String
age	Int

Data Values

Actual values

name	age
Natasha	34
Mark	51

Data Objects

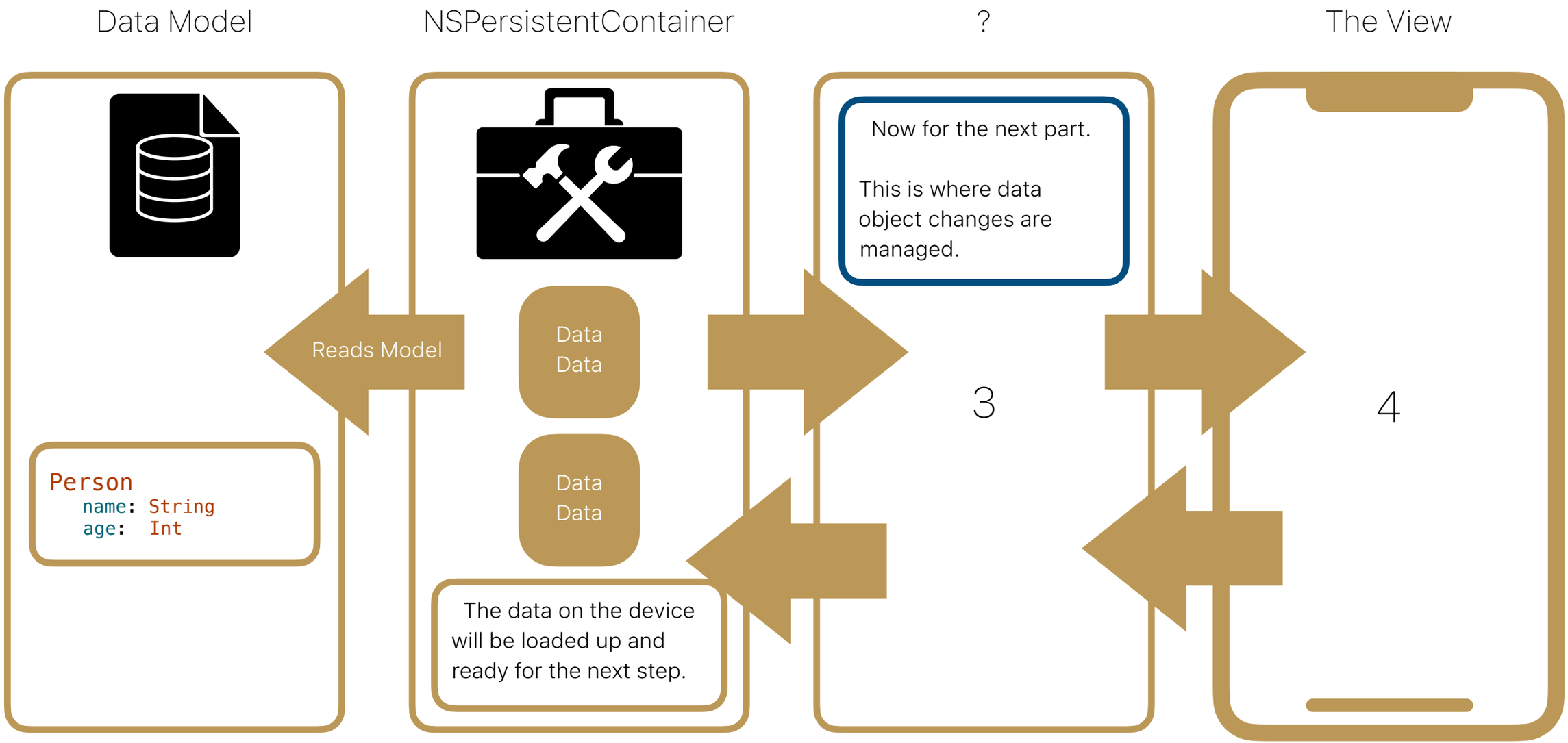
```
class Person {
    var name = “Natasha”
    var age = 34
}

class Person {
    var name = “Mark”
    var age = 51
}
```

Why isn’t this part in the flow?

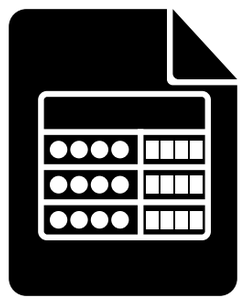
I intentionally left out the data values from the flow because you don’t have to do anything to get it working.

Part 2 in The Flow: NSPersistentContainer

When you start your app, one of the first things you want to do is instantiate your NSPersistentContainer so it can read data model and then your persisted data and get it ready.



3. Managing Data Objects

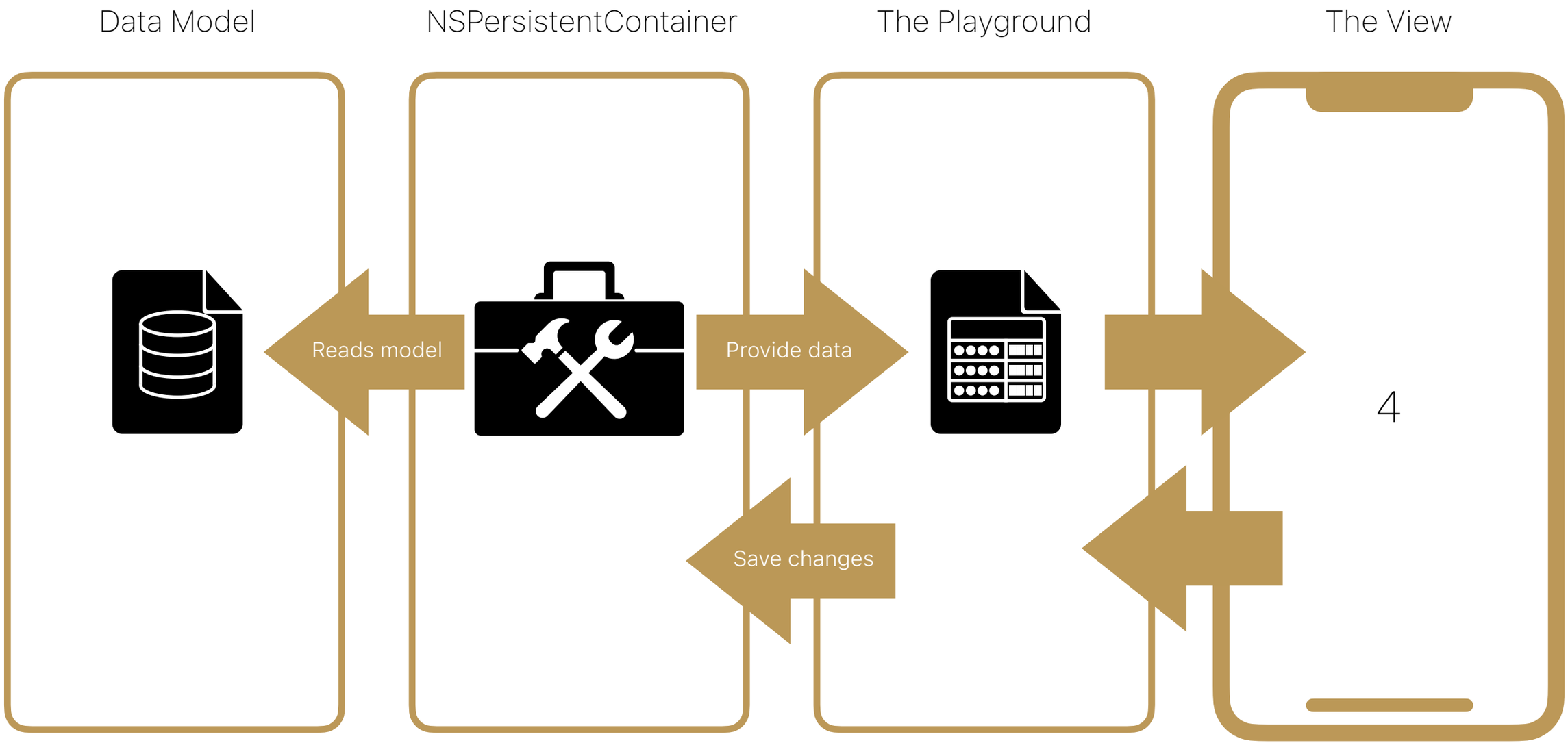


Once Core Data understands the data models and where the data is stored, it can retrieve and give data objects to your app.

At the end of this section, you will understand more about how and where data objects are managed.

The Playground - The Third Part

We need a playground where we can play with data. Many times, after fetching data, you enable the user to insert new data, change the data, delete the data, or maybe undo changes. This is where your users will do it, where the data objects (the things with actual values) are managed.



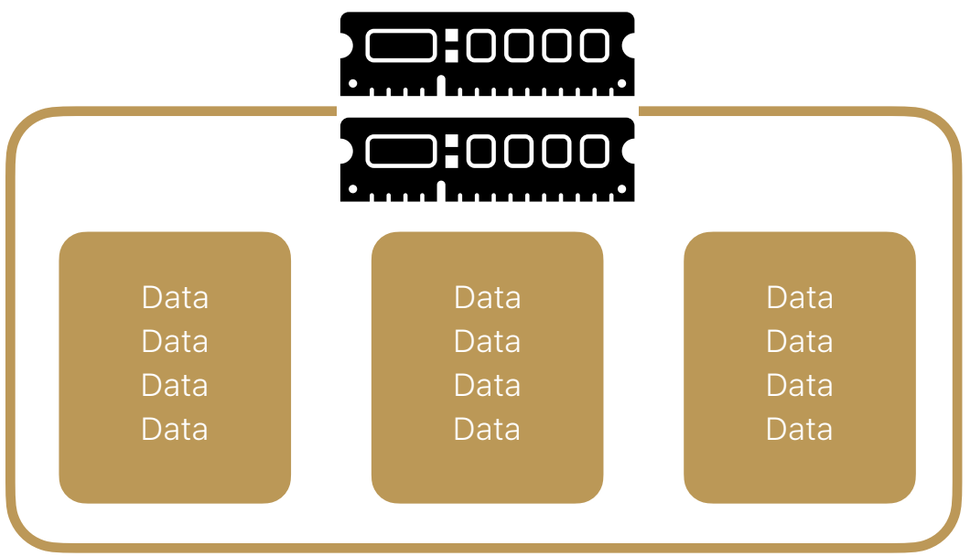
Memory

When any application works with data or data objects, it is usually done in memory or RAM. “RAM” is short for “random access memory”.

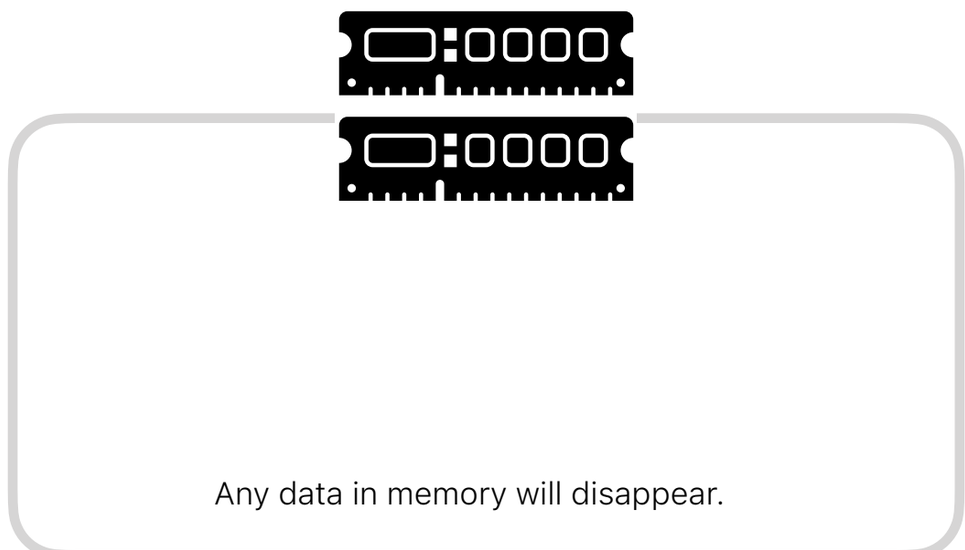
Think of RAM or memory as temporary. If the device loses power, all the data that is in memory is gone.

The operating system gives your app memory to work within. When you close the app, the operating system reclaims it and any data within is gone.

Memory with Power



Memory with no Power

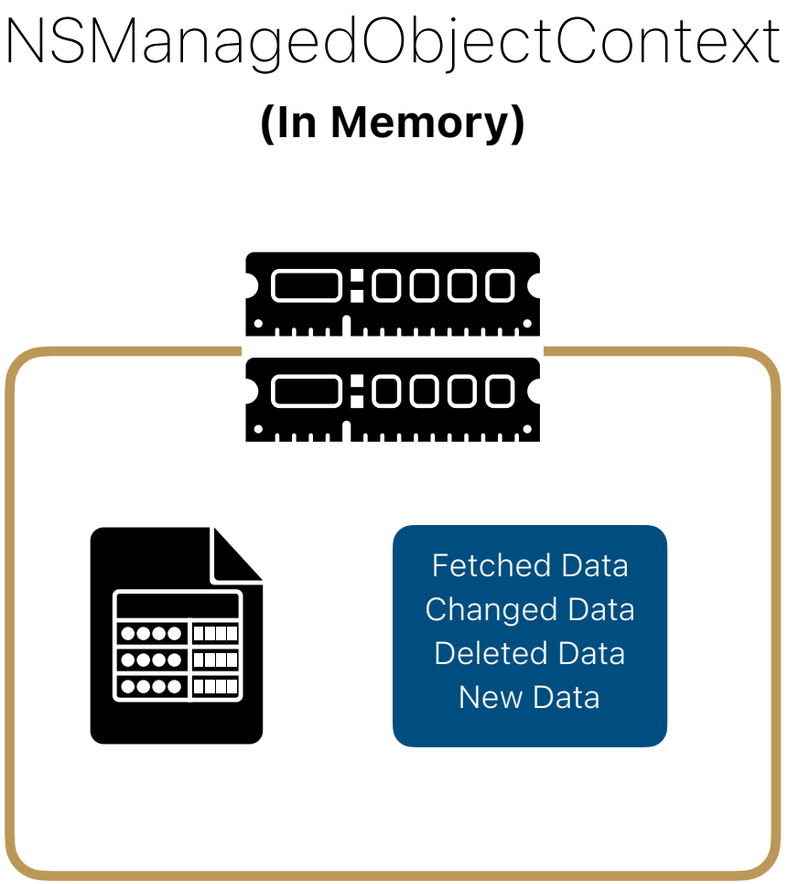


Core Data will use memory to temporarily store data changes. This is useful in case the user changes their mind and decides not to save or delete data (undo).
It’s also useful because memory is **super fast**.

NSManagedObjectContext

Core Data uses the NSManagedObjectContext as a space to let you freely change data in memory. It is where updates, deletions, and the creation of new data objects take place.

You manage objects here. (Remember, data **objects** are data models that have values.)



The NSManagedObjectContext:

- Enables you to fetch specific data objects from persistent storage
- Manages object changes in memory until written to the persistent store
- Allows you to undo changes
- Has the ability to perform data object work in the background

All of these data object changes are just in memory.

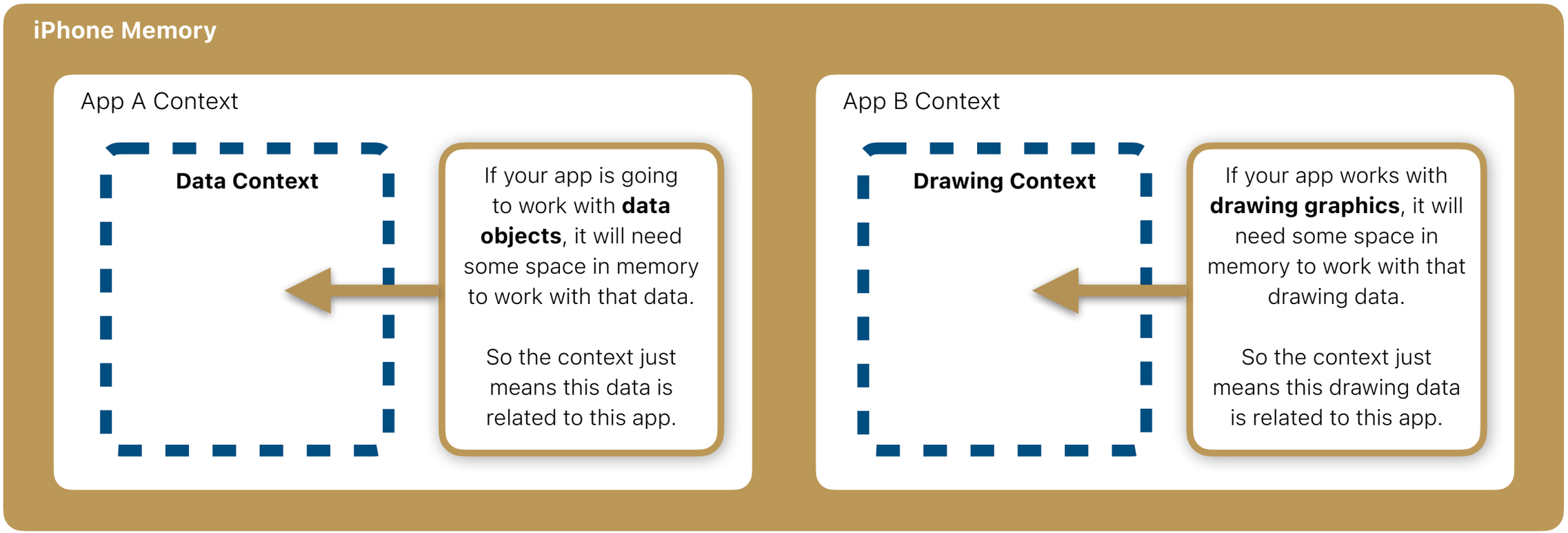
You will have to call a function to have NSPersistentContainer commit (save) the changes to the persistent store.

Managed Object..."Context"

The word **context** means "everything related to some event, or idea so you can make sense with what is being said or shown".

When you have data objects in memory, the operating system has to know what this memory is related to. Is it for App A or App B?

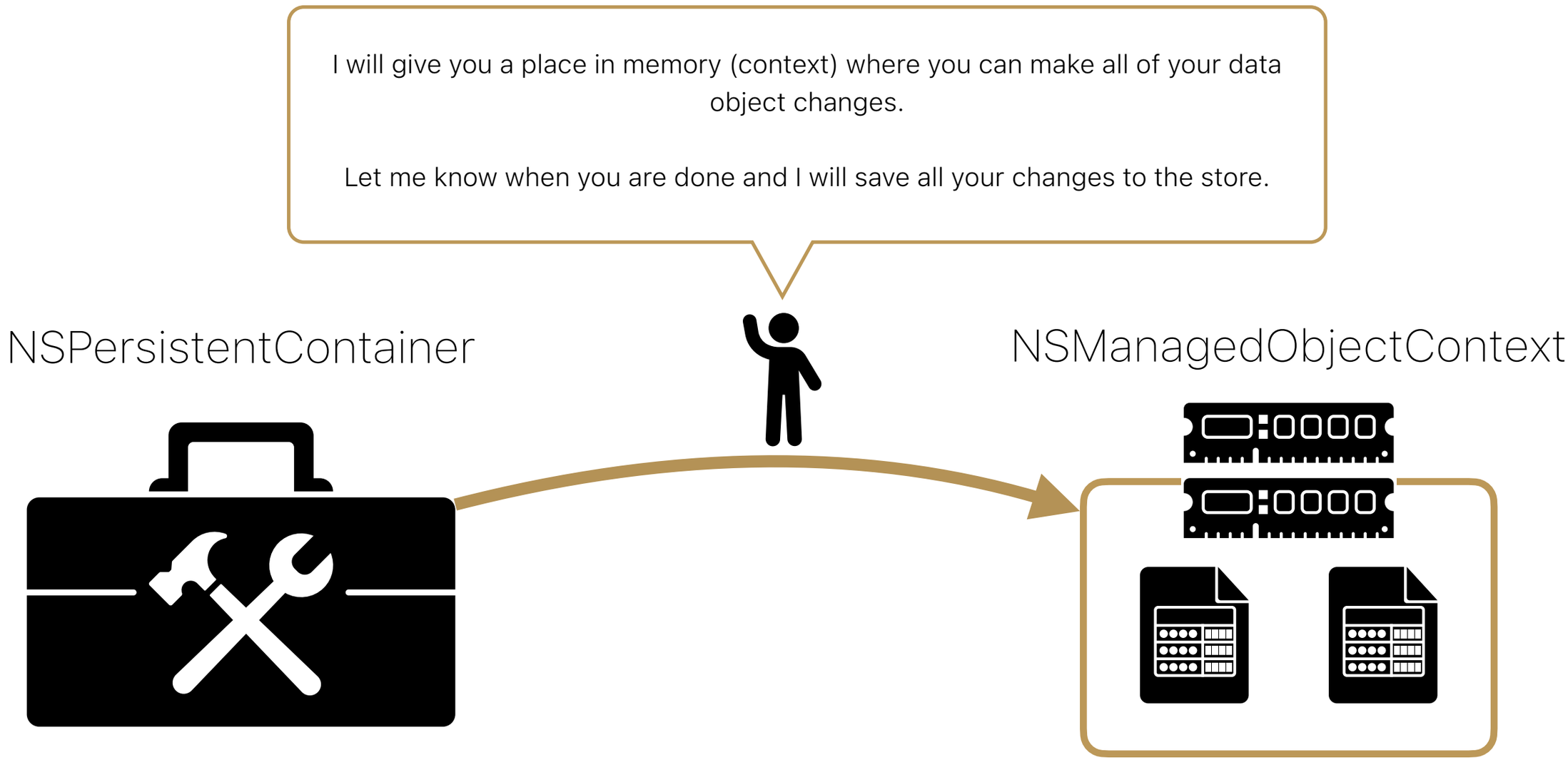
So context lets people and computers know how the things in memory are connected with which app and what they are used for. One app can have many contexts being used for different purposes.



Note: There are many contexts for many frameworks and technologies within the available libraries used for developing apps. Here, you see examples of data and drawing contexts. **When you see "context", think "space in memory to work within".**

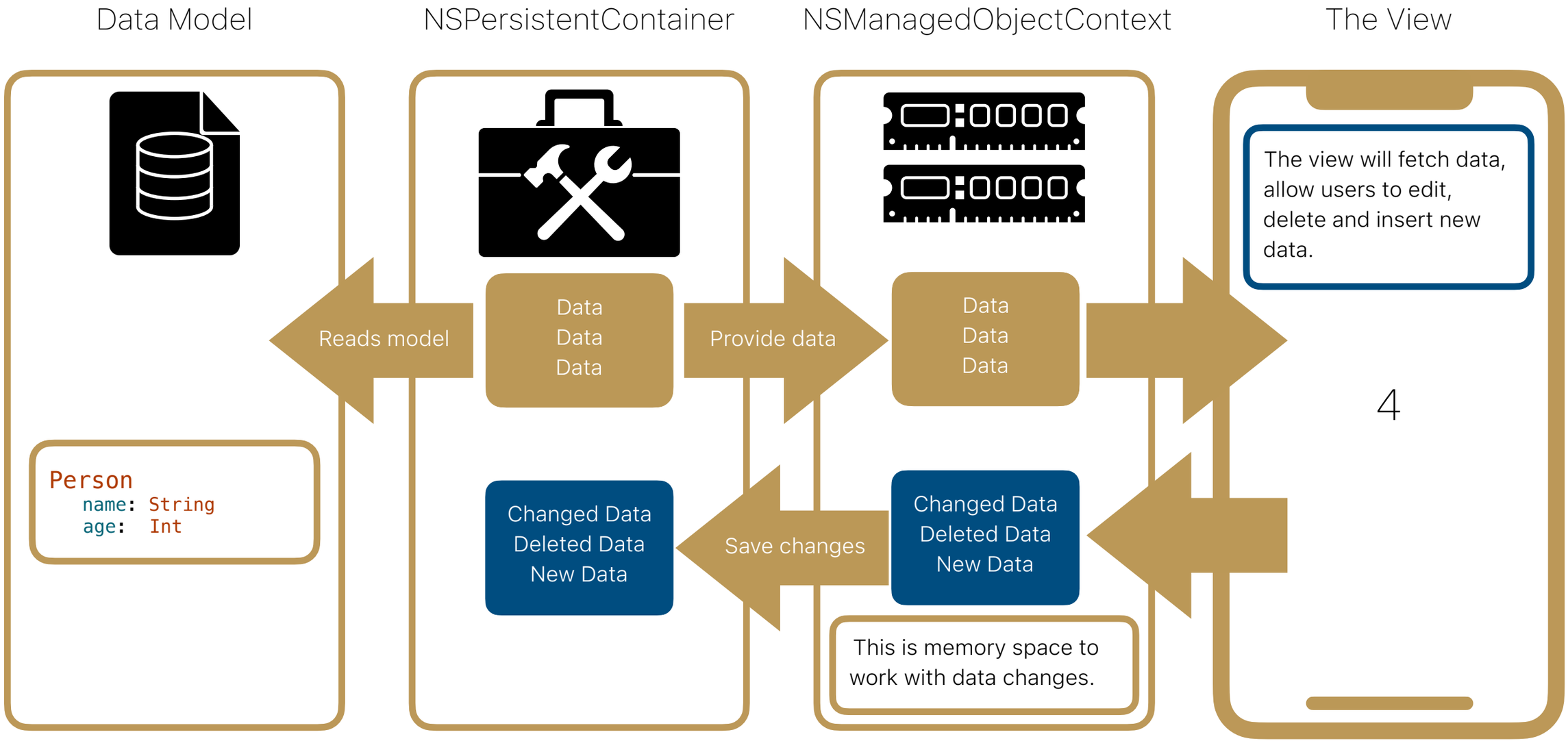
NSManagedObjectContext is from the NSPersistentContainer

The NSManagedObjectContext is one of the tools provided to you from the NSPersistentContainer.



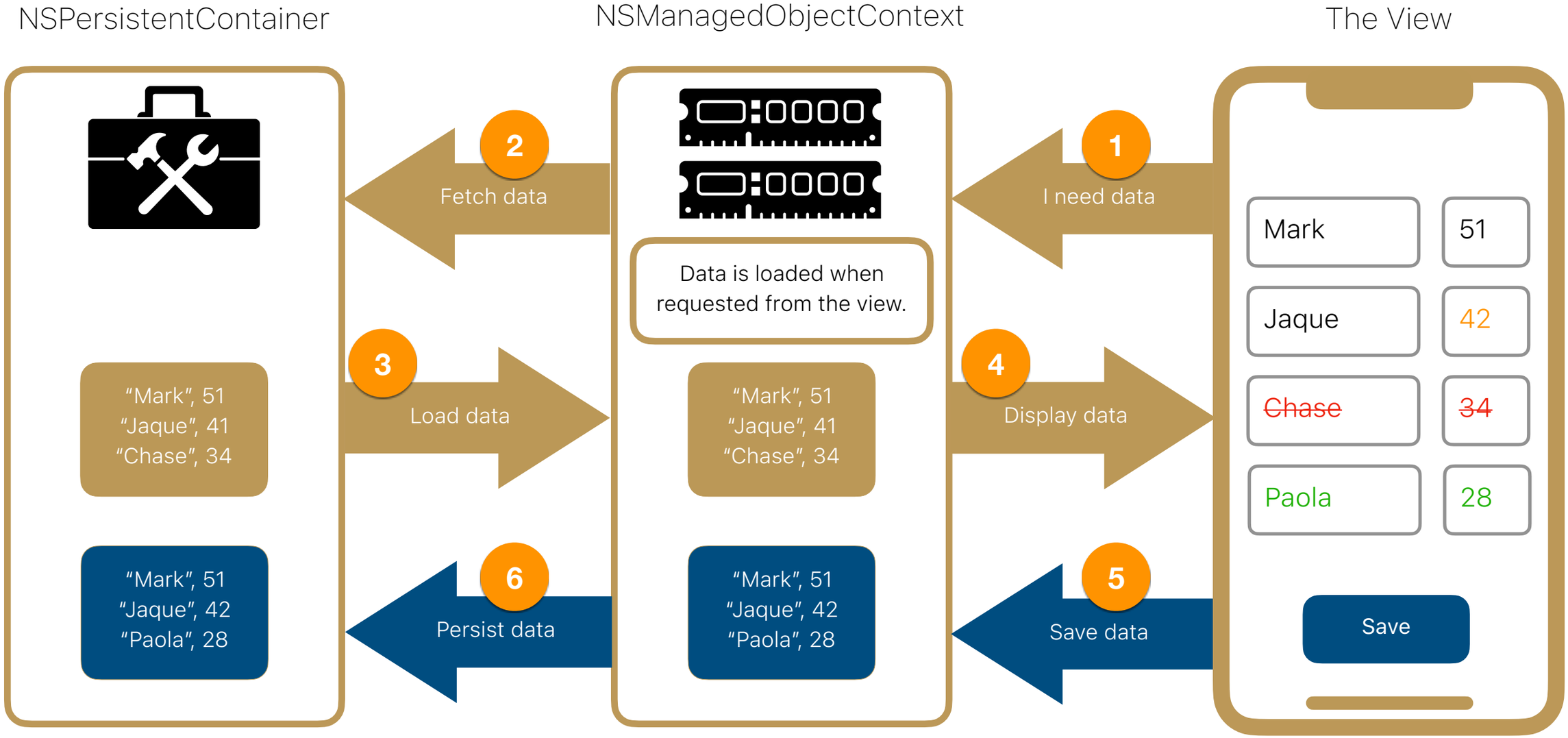
Part 3 in The Flow: NSManagedObjectContext

The **NSManagedObjectContext** is provided to you by the NSPersistentContainer. It's giving you a space in memory (context) to work with your data objects before committing it (saving it) to persistent storage.

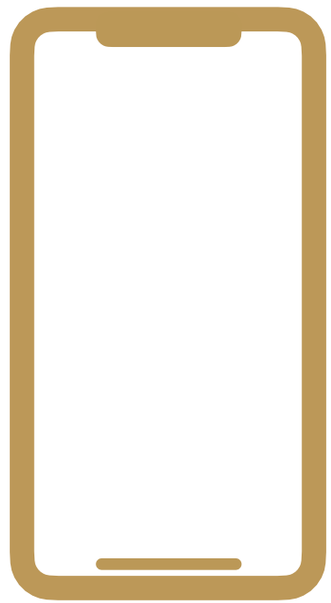


Managed Object Context and Data Flow

The NSManagedObjectContext is not automatically filled up with all the data from your persistent store. Think of it as a temporary place where you can fetch, create, change and destroy things within it. So only when requested will the persistent container load data into the managed object context.



4. The View

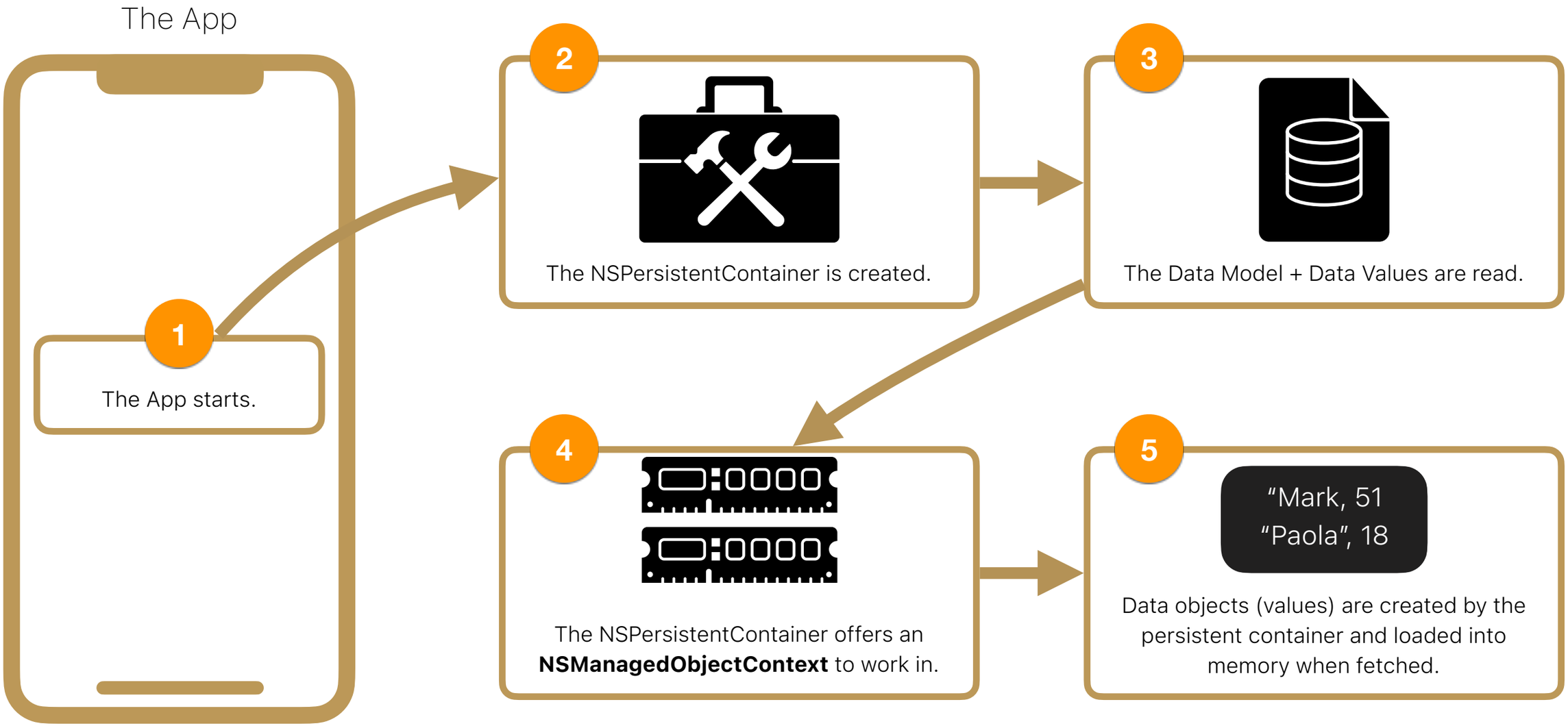


When your app starts there is a particular order in which it initializes Core Data for use.

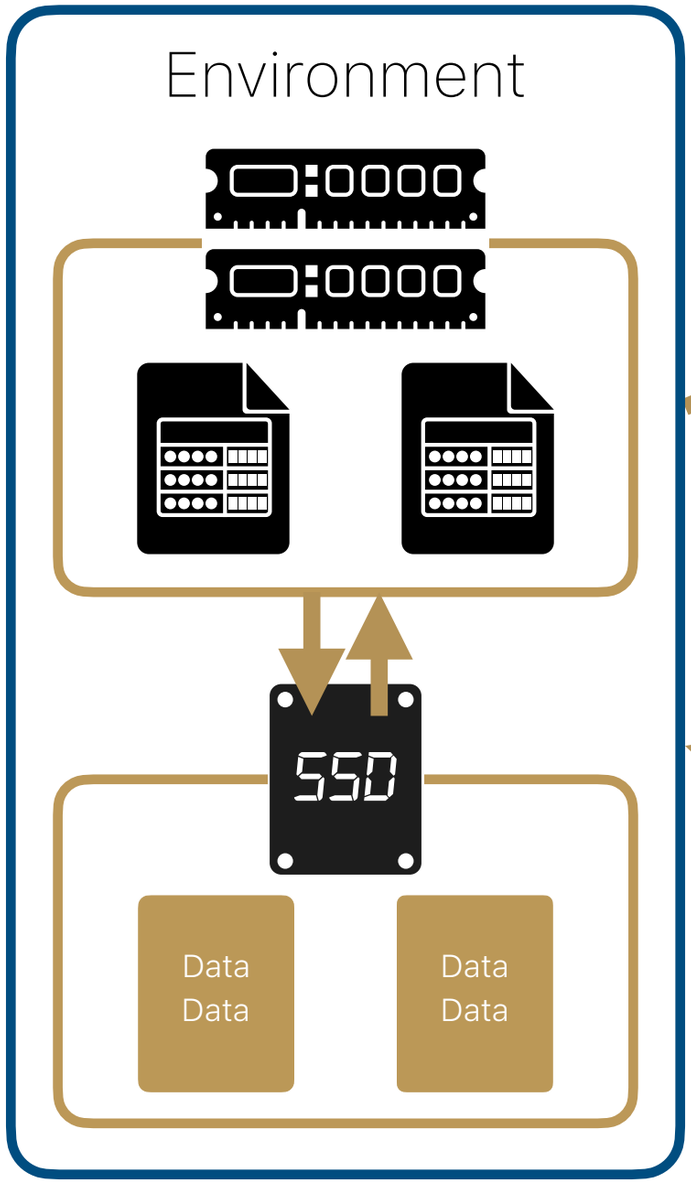
You will learn this order and the idea of how views work with Core Data in this section.

Loading Core Data

The last piece of the puzzle is the app itself. When the app starts, it will create the **NSPersistentContainer** first. This is a simplified idea of how it all works so you can think with it. You will learn more details about each of these parts in this book.



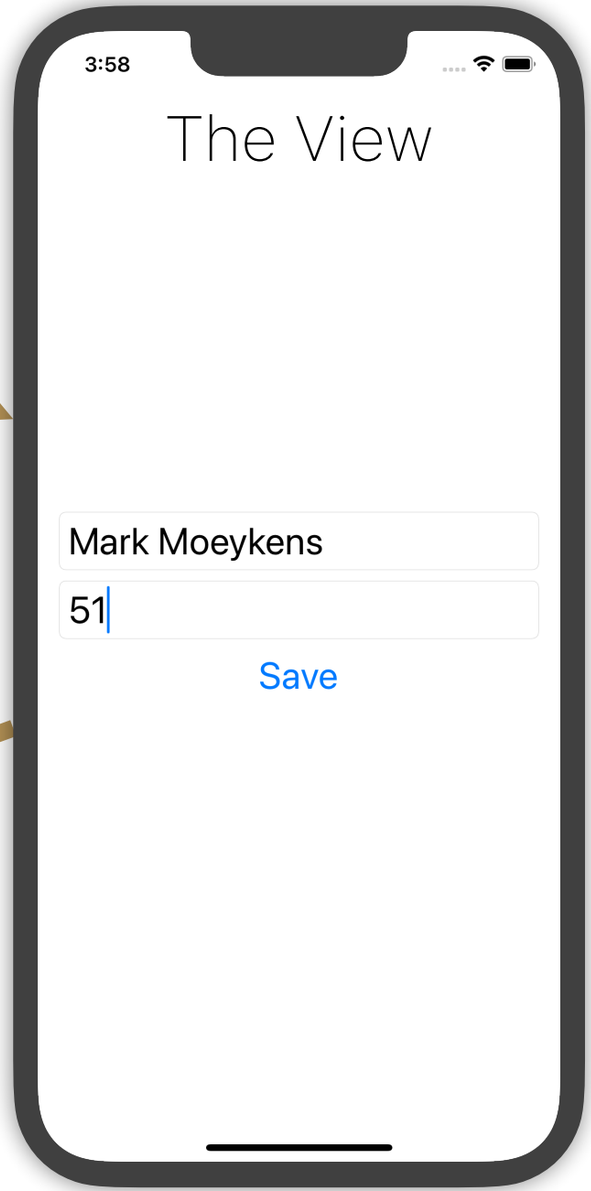
Using @Environment



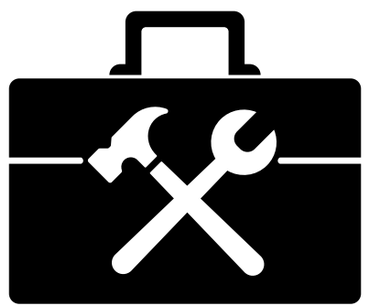
A common practice in SwiftUI apps is to store the NSManagedObjectContext in the **environment**.
This way it is available to all views so you can fetch and save data.

SwiftUI has some super easy ways to get data and display it.

Saving your changes is also super easy.



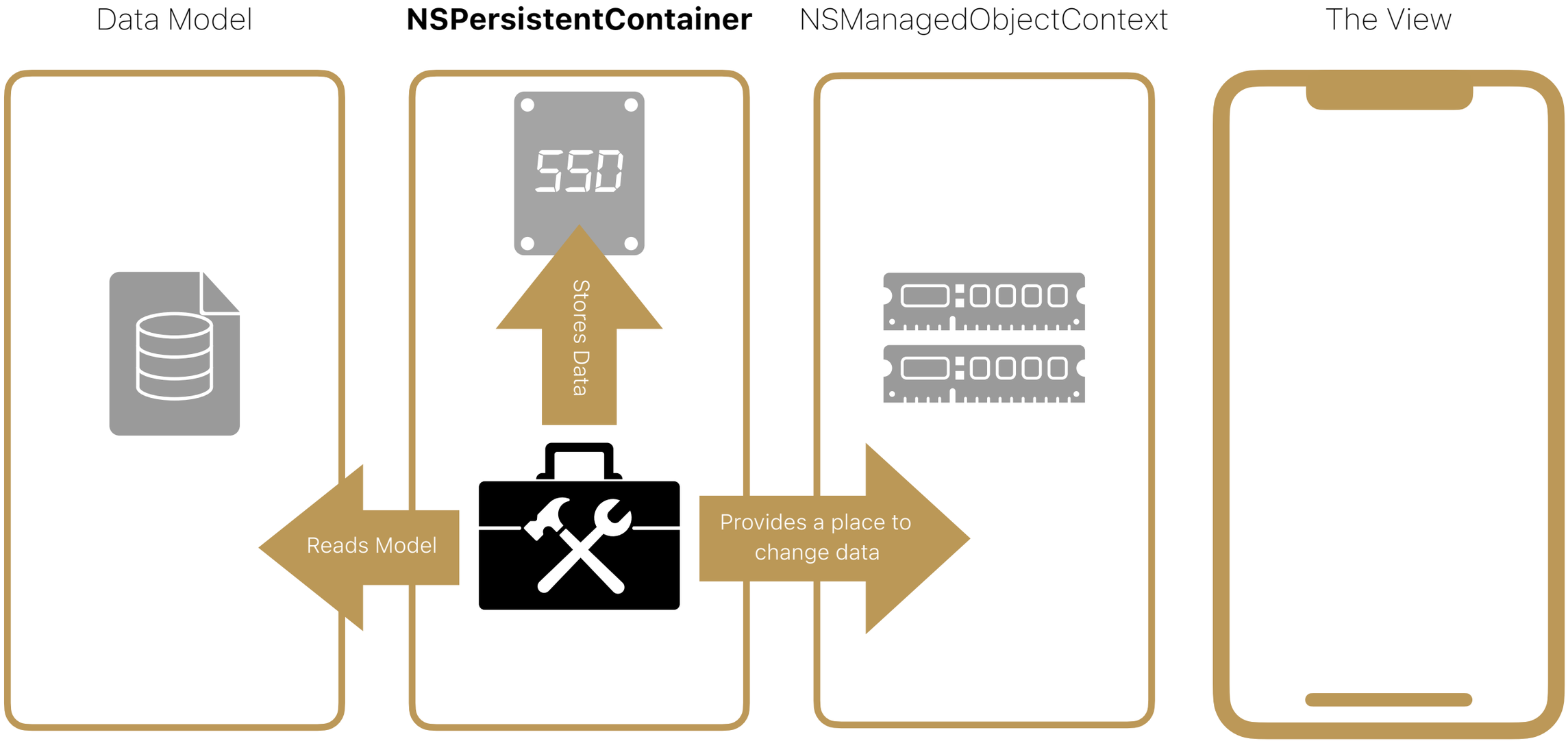
Summary



You've learned a lot of concepts around Core Data. You'll be using these concepts all throughout the book. So let's summarize them.

The Main Part of Core Data

If you were to ask me what is the main part that makes Core Data work, I would say it is the **NSPersistentContainer**. Everything you do in Core Data revolves around this one main part. Over the years, Apple has made this the Core Data central toolbox for everything you need.



The Flow

Each of the 4 parts all have their own responsibilities.

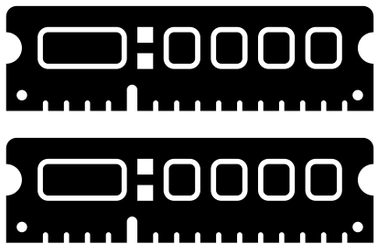
Data Model

Use the Data Model to describe what your data looks like.

NSPersistentContainer

The persistent container will read your data model and then load your data from the persistent store.

NSManagedObjectContext



This space (context) is used to manage your data objects.

It will track changes, deletions, insertions, etc.

It provides you the tools to undo or commit your changes to the persistent store.

The View

The app will create the NSPersistentContainer and load data from the persistent store.

The views will access the managed object context from a central place.

FIRST EXAMPLE

In the previous pages, you learned about the 3 main tools the app needs to work with Core Data. You should have a pretty good concept of what they are and how the app uses them.

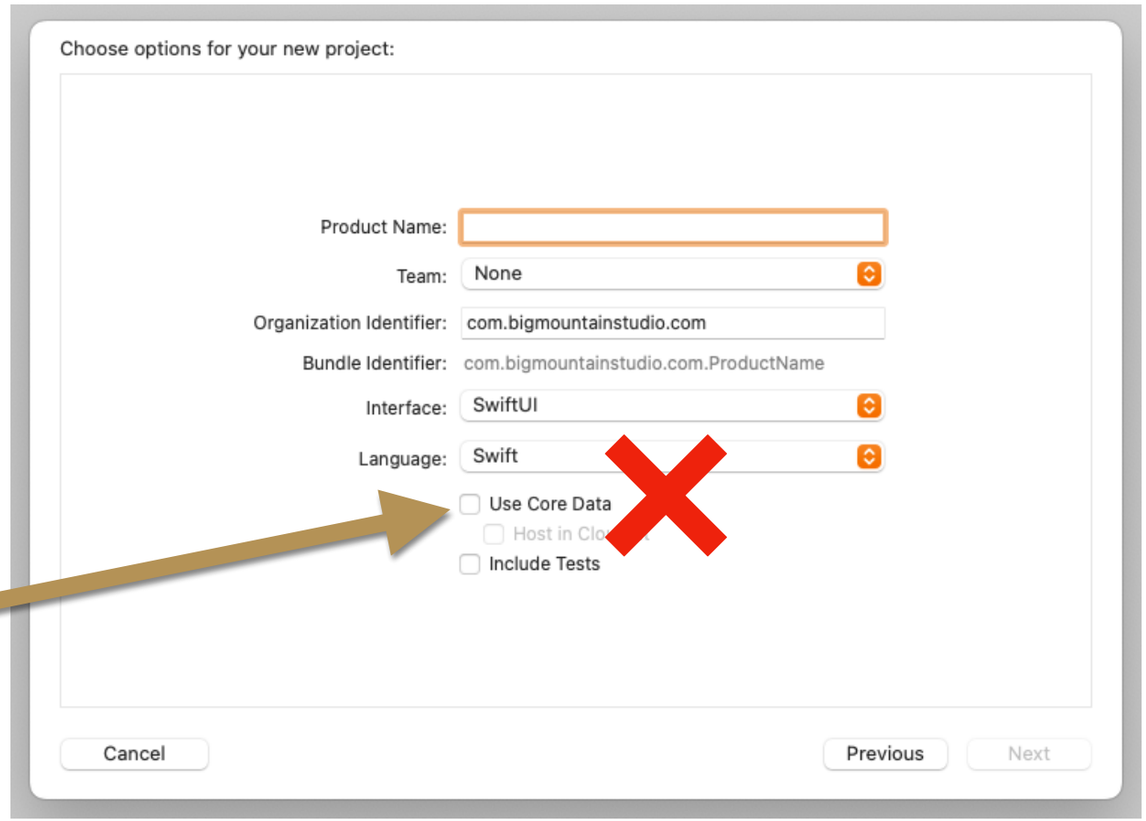
Now you're going to learn what those concepts look like in code as we set up a data model, add some data and show it in a SwiftUI view.

1. Setting Up the Data Model

By the end of this section, you will learn the basics of creating and setting up a data model in Xcode.

New or Existing Project?

This example will take you through creating your first example with Core Data. You can use it on an existing project or a new project.



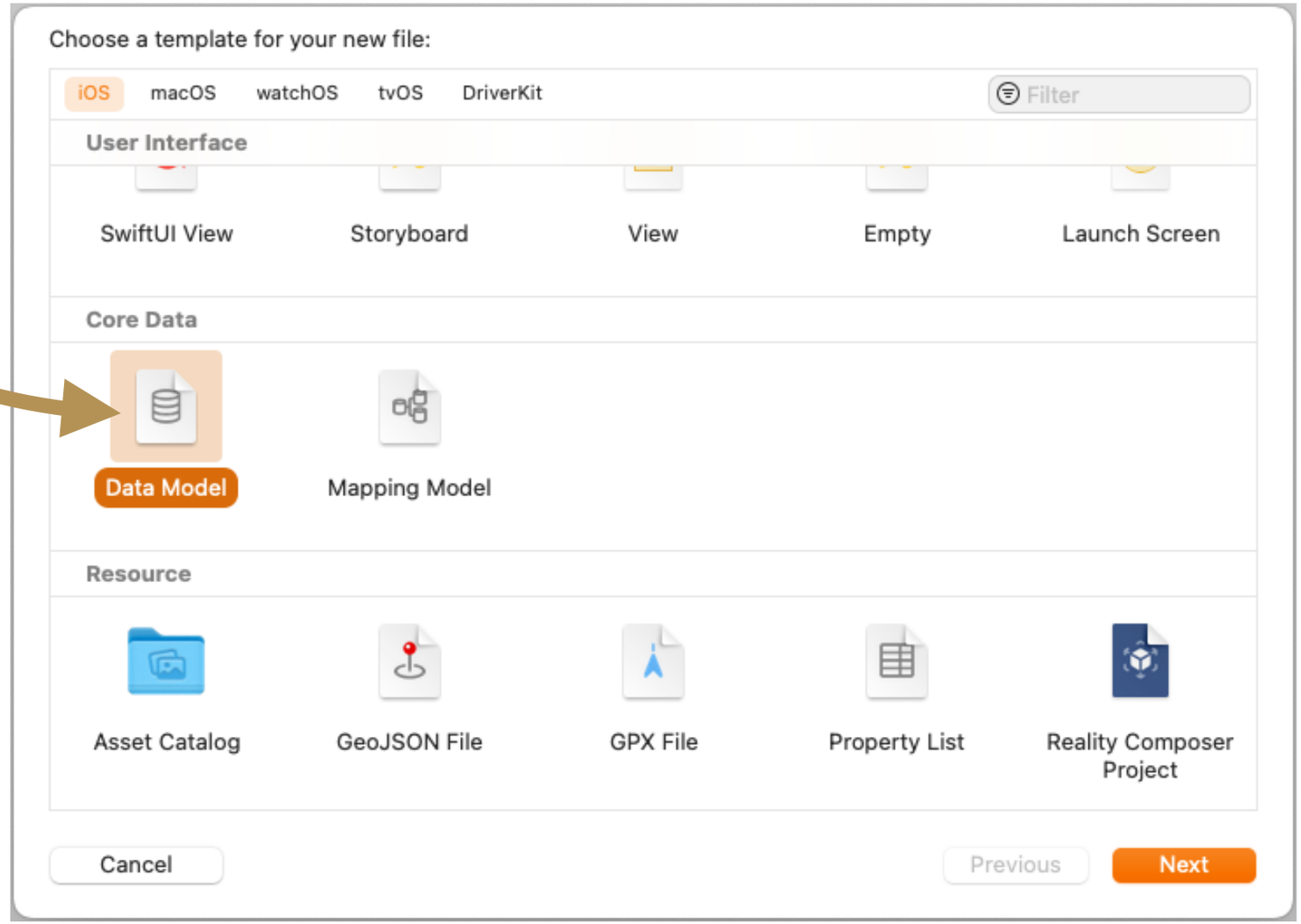
If you create a new project, **uncheck** the "**Use Core Data**" option.

This option will autogenerate a lot of code that can be confusing when first starting out.

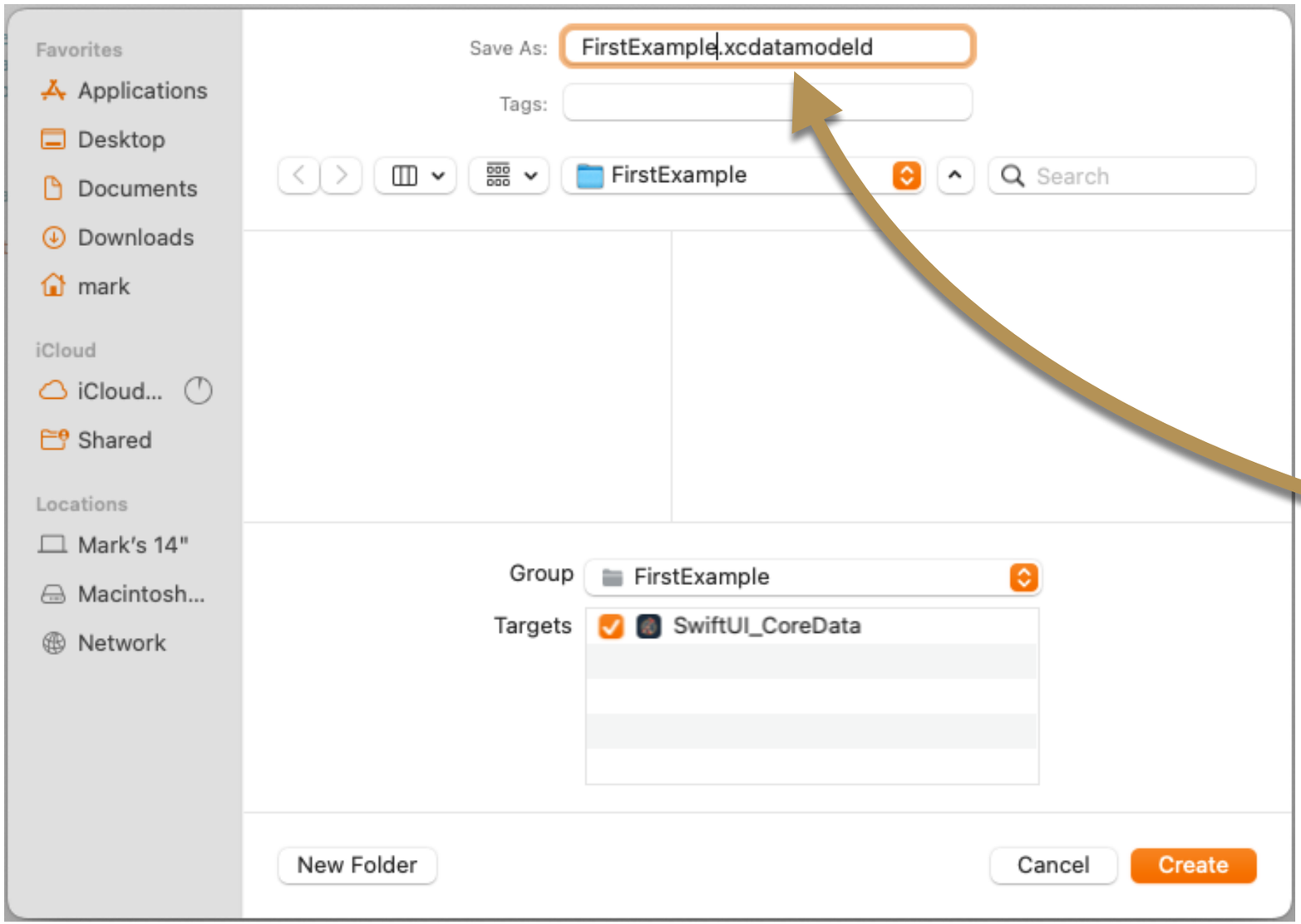
We're going to make it very simple.

Create a New Data Model

When you add a new file to your Xcode project you will want to scroll down to the Core Data section and select **Data Model**.



The Data Model Name



On the next screen, give the data model any name you want.

Let's break down the file extension.

xc - Xcode

datamodel - You know from the previous chapter that a data model just describes what the data looks like. It's not actual data values or data objects.

d - Yes, there is a "d" at the end. What is this for? It stands for "directory". The directory is structured like this:

- FirstExample.xcdatamodel**d**
  - FirstExample.xcdatamodel (no "d")
    - contents

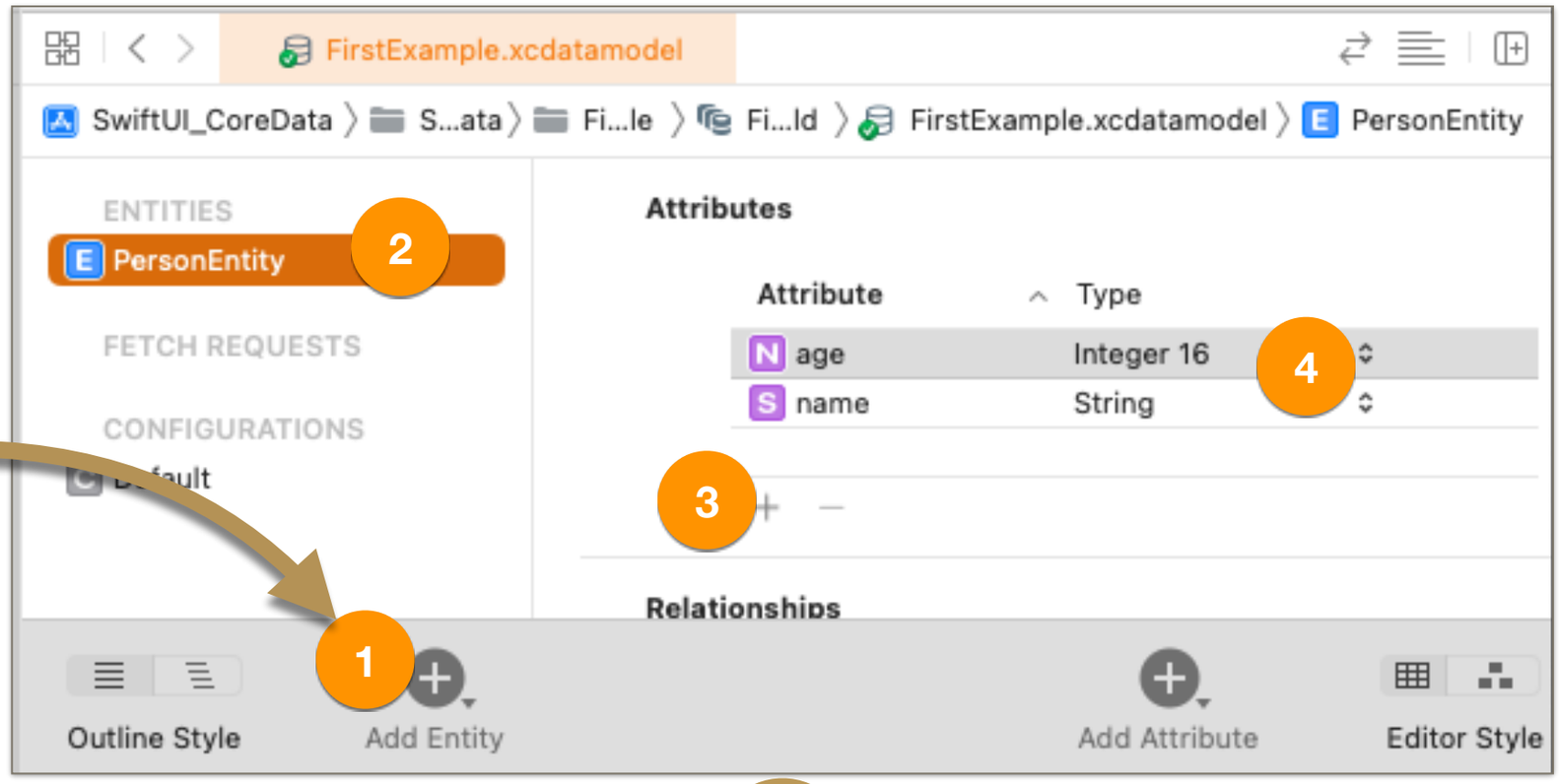
The contents file is an XML file representation of the data model.

Data Model Editor

We're going to recreate this data model which you saw earlier using the editor:

```
struct Person {
    let name: String
    let age: Int
}
```

1. Click the Add Entity button
2. Give your entity a name
3. Click the + button in the Attributes section
4. Add a name and assign a type



The editor will force you to make the first letter in entity names **uppercase**. It will also force you to make attribute names start with a **lowercase** letter.

Think of entities as classes and attributes as properties.

Can I change the position of attribute names?

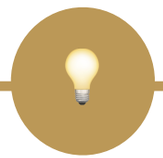
No, they are sorted alphabetically. Sometimes you will notice they are in random order when you first add them. But if you close this file and open it again, the attributes are sorted alphabetically again.

Attribute Types

You probably noticed that the available attribute types didn't have a plain **`Int`** as we used in our struct.

Numbers can be broken down into how big the number can get or the range of values they offer.

This table will help you decide which types to use.

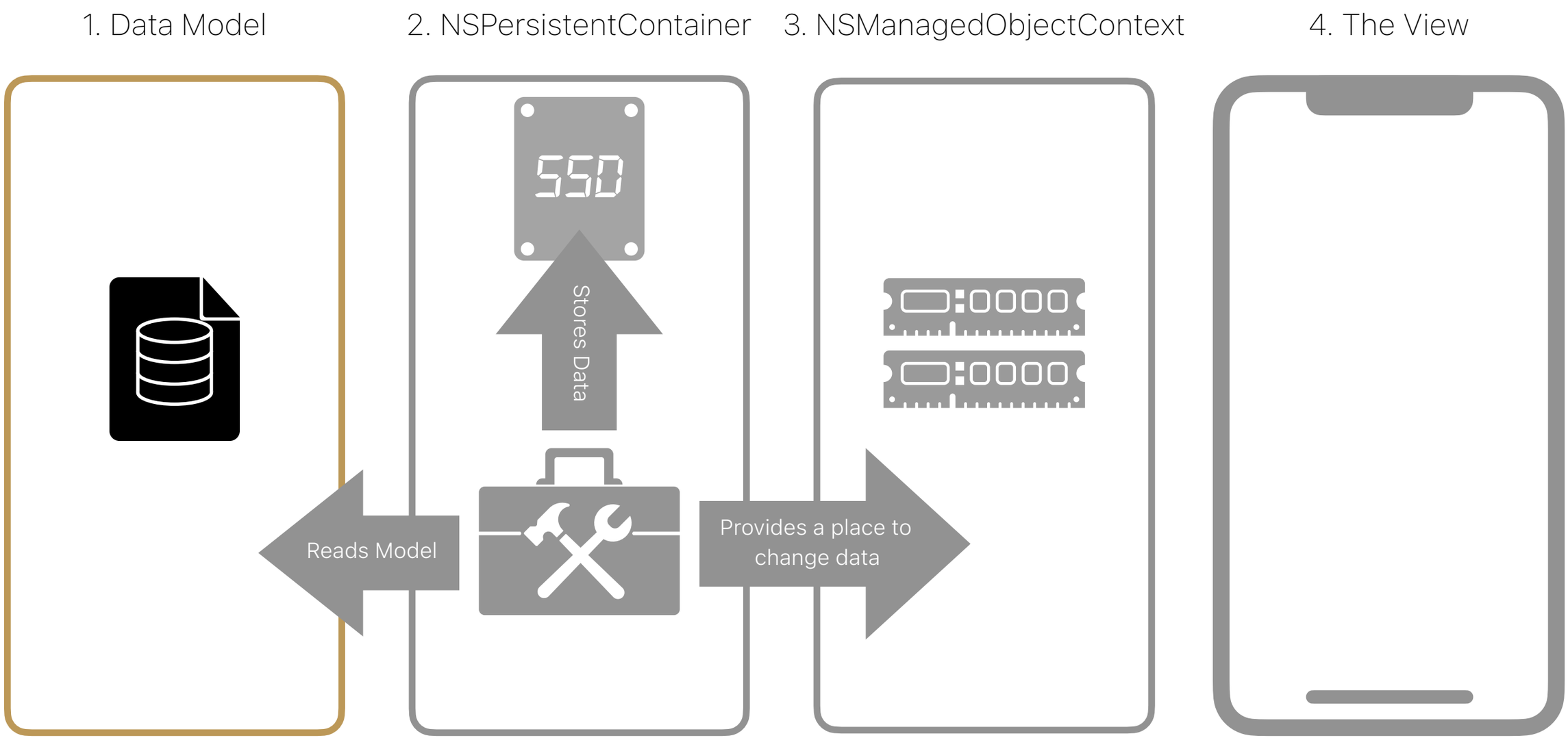


Note: A Swift Int type can be Int32 or Int64 based on whether the platform is a 32 or 64-bit system. That's why you see it next to both Int32 and Int64. (Most are 64-bit now.)

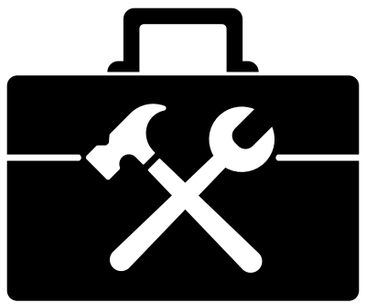
Attribute Type	Swift Type	General Information
Binary Data	Data	When you want to store images, videos, sound files, or any other data objects
Boolean	Bool	True or false
Date	Date	Dates and times
Decimal	Decimal	Highest level of precision. Great for currencies.
Double	Double	More precise than a Float (at least 15 decimal places)
Float	Float	A number with a decimal when you don't need the greatest precision
Integer 16	Int16	When you need a smaller number (-32,768 to 32,767)
Integer 32	Int32, Int	When you need a bigger number range (-2,147,483,648 to 2,147,483,647)
Integer 64	Int64, Int	For HUGE numbers (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
String	String	Text
Transformable	NSObject	When no other type will do
Undefined		Default type that will give you an error to remind you to set a defined type
URI	URL	Web, file, or any other URL formats
UUID	UUID	When you need a unique identifier (id)

First Step Done

You added a data model file to your project and added an entity with some properties (attributes). Step one is now complete.



2. The Persistent Container



It is time to set up your persistent container.

A common practice is to create your persistent container when your app starts or when you first need the data.

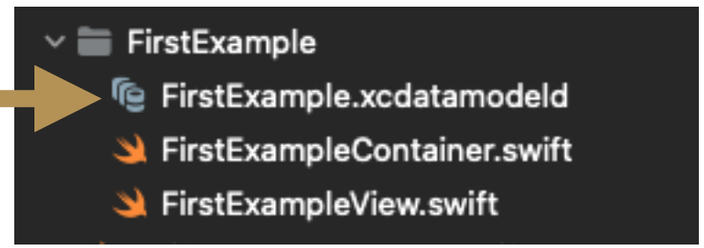
And you probably want to keep it alive for the life of your app (or the life of the views that need it) so you don't have to keep creating it.

The Persistent Container

Starting as simple as possible, the only responsibility of this class will be to create our persistent container.

```
import CoreData

class FirstExampleContainer {

    let persistentContainer: NSPersistentContainer


    init() {

        persistentContainer = NSPersistentContainer(name: "FirstExample")

        persistentContainer.loadPersistentStores { _, _ in }

    }
}
```

When the persistent container is created, it needs the name of the data model to read.
The name should be exactly the same name as the file name.



Xcode Project Navigator

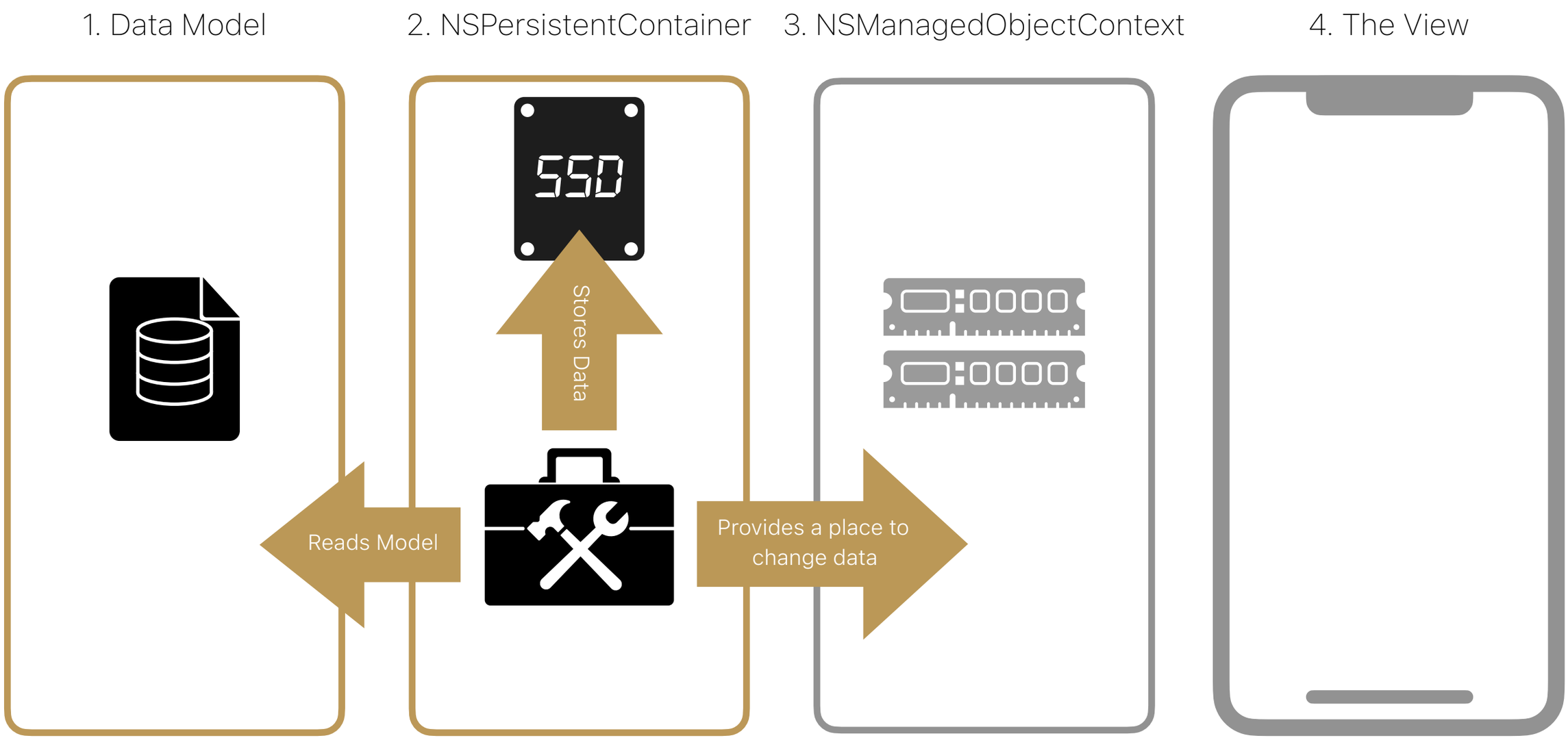
Now that the persistent container knows what the data should look like, we can load the persistent stores where the data exists.

It knows exactly where to look and get it.

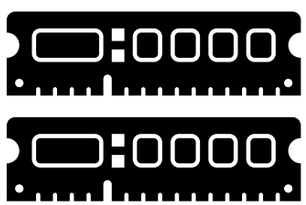
The closure and the parameter names are intentionally left blank. This is where you check for and handle any errors. We will cover this later in the book.

2 Steps Done

We have our data model defined and we created our persistent container. The next step is to set up the managed object context in a way that our views can use it.



3. The Managed Object Context



We need to set up a space in memory (context) to manage our data objects. SwiftUI has a really easy way in which we can do this.

Managed Object Context Environment

In your project's App file, you will want to use the environment modifier on your view to attach the managed object context.

```
import SwiftUI

@main
struct SwiftUI_CoreDataApp: App {
    var body: some Scene {
        WindowGroup {
            FirstExampleView()
                .environment(\.managedObjectContext, FirstExampleContainer().persistentContainer.viewContext)
        }
    }
}
```

The environment modifier provides a special property just for your managed object context.

Remember, `persistentContainer` is the property we created to hold our `NSPersistentContainer`.

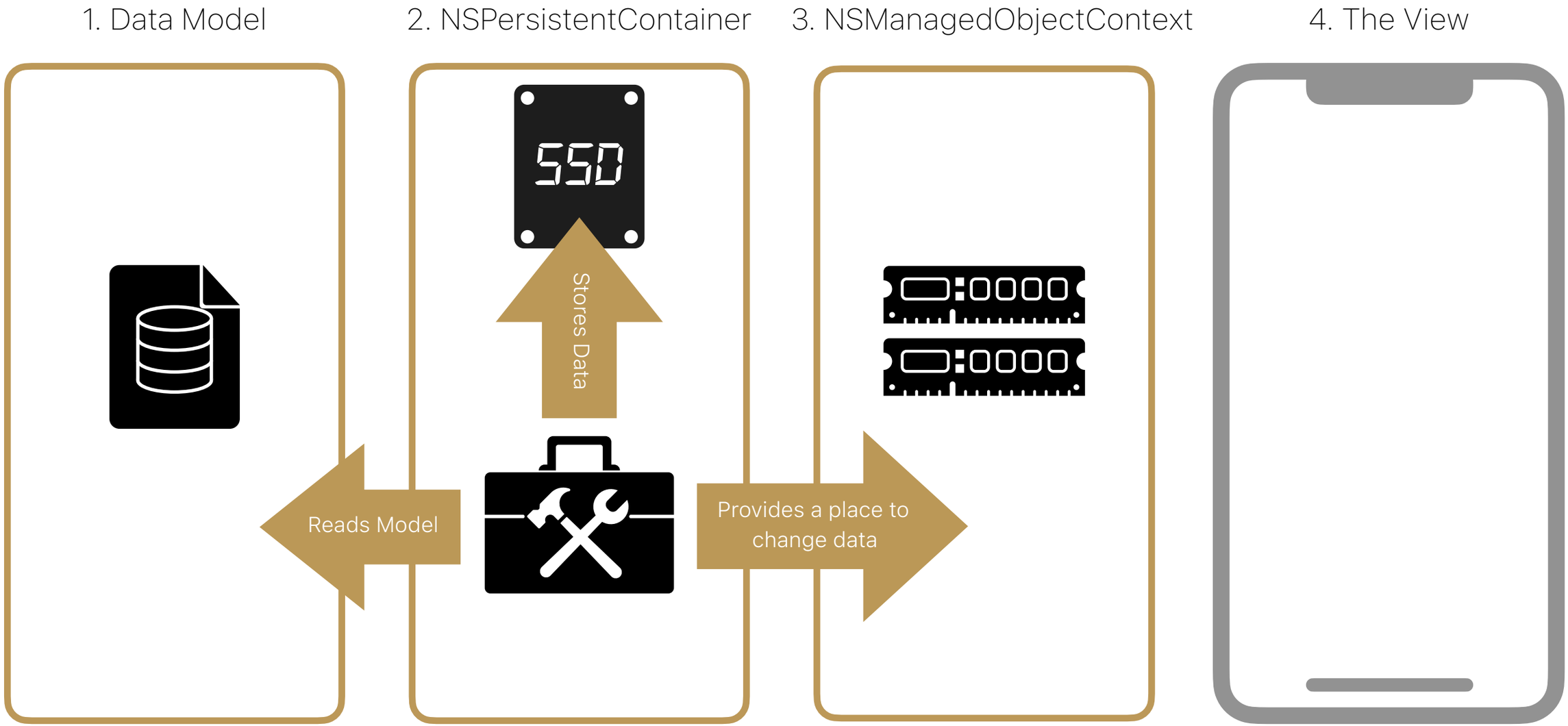
Persistent containers give us a managed object context (playground in memory) to make changes to data.

Note: The way environment modifiers work is they make an object available to the view it is attached to AND all child views from there. This provides views with a way to fetch data, make changes, and save data.

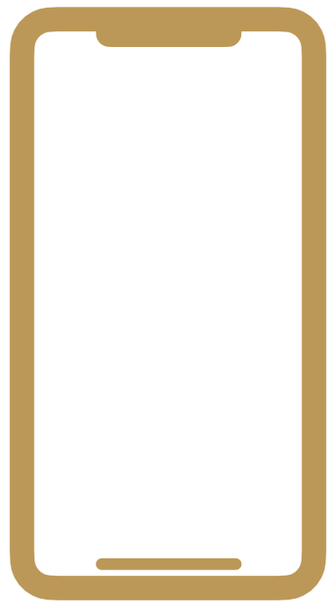
More Info: If you want to learn more about how the environment works, take a look at the Environment chapter in the "Working with Data in SwiftUI" book.

3 Steps Done

The persistent container is created and the `viewContext` (managed object context) is added to the environment. Now the view has to access the managed object context so it can get some data and display it on the screen.



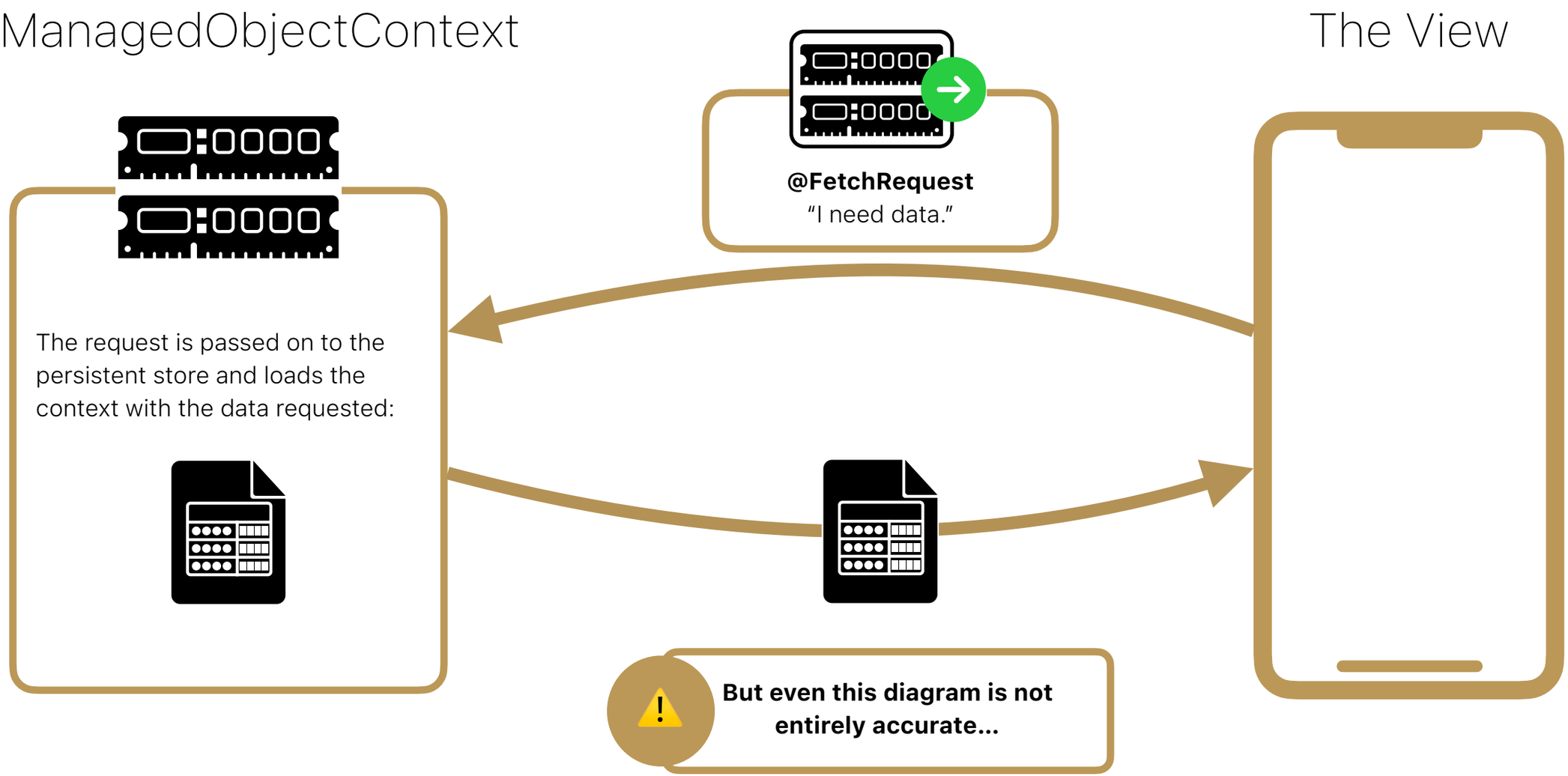
4. The View



With Core Data set up, it is now time to start working with data objects.

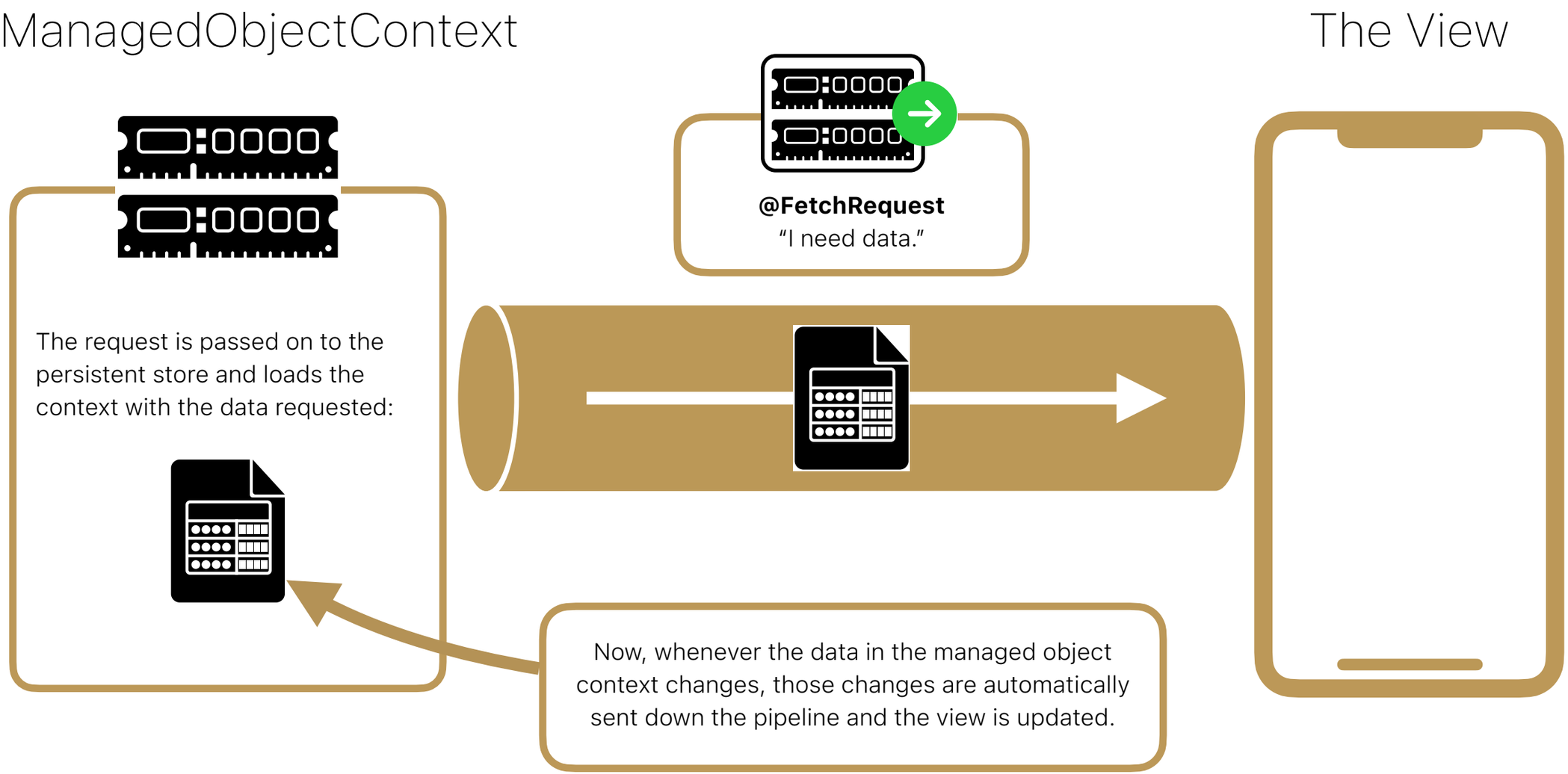
@FetchRequest Concept

The view is going to use a property wrapper called **@FetchRequest** to retrieve data from the managed object context. The request is passed from the view to the managed object context.



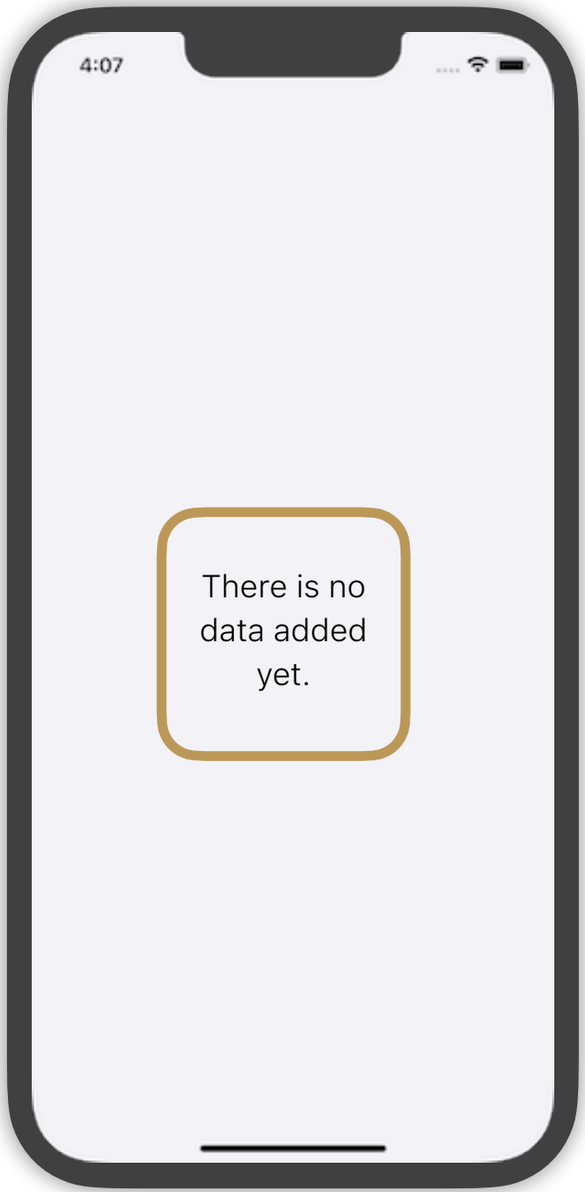
@FetchRequest Pipeline Concept

The previous diagram gives you the idea a fetch is made, data is returned and that is it. But really, when the @FetchRequest is used it is almost setting up a pipeline. So whenever data changes in the managed object context **the view automatically gets updated**.

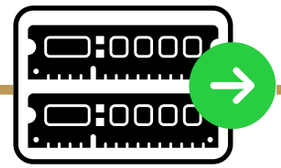


@FetchRequest

Here's your first view using Core Data. It's not very exciting. We will add data on the **next page**.




```
struct FirstExampleView: View {
    @FetchRequest(sortDescriptors: []) private var people: FetchedResults<PersonEntity>

    var body: some View {
        VStack {
            List(people) { person in
                Text(person.name ?? "")
            }
        }
        .font(.title)
    }
}
```



The **@FetchRequest** property wrapper sets up that pipeline to the managed object context.

sortDescriptors - This will sort the data that you get back in the way you specify. It is an array, meaning you can provide many sorting options. It's left blank so the data will be unsorted.

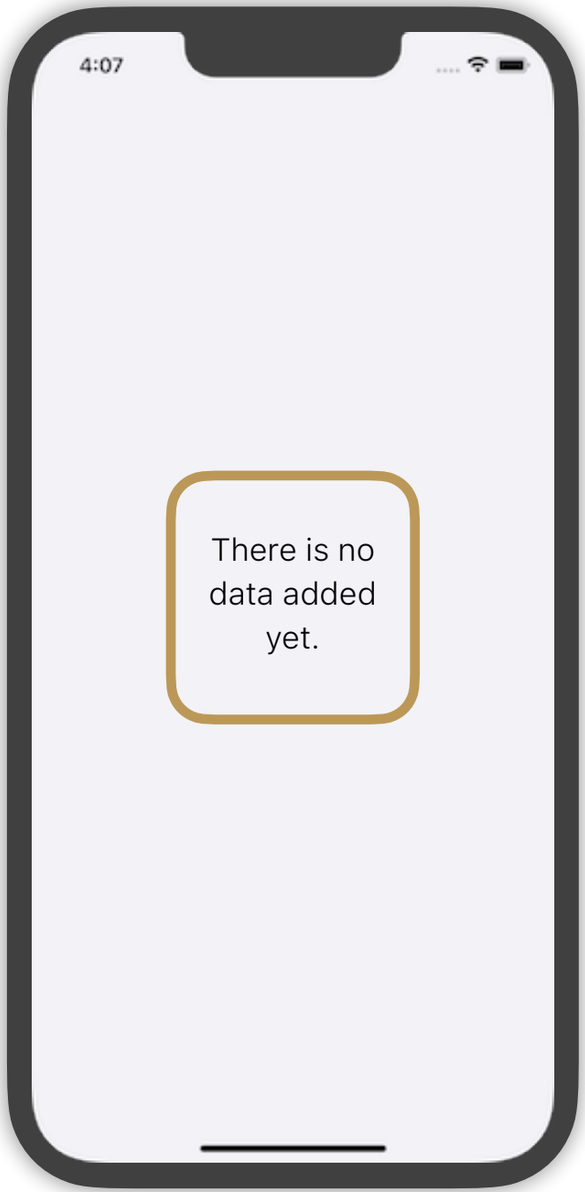
FetchedResults - This is the collection of your data objects that were retrieved from the store. Notice it is generic so you set the type of data you want returned within the brackets.

What's amazing about this code is:

- Xcode synthesized (autogenerated) that `PersonEntity` class for you just by looking at the Data Model. Autocomplete works with it.
- The type you set on FetchedResults (`PersonEntity`) is literally all you have to do so Core Data knows what data you want. It's that simple!

No ID?

How come there is no id specified for the List view?




```
struct FirstExampleView: View {
    @FetchRequest(sortDescriptors: []) private var people: FetchedResults<PersonEntity>

    var body: some View {
        VStack {
            List(people) { person in
                Text(person.name ?? "")
            }
        }
        .font(.title)
    }
}
```

The person's name is an optional string. Core Data defaults many of the types to optionals.

One way to handle this is with a nil coalescing operator (??).

When you use List, you sometimes have to specify a key path to an id.

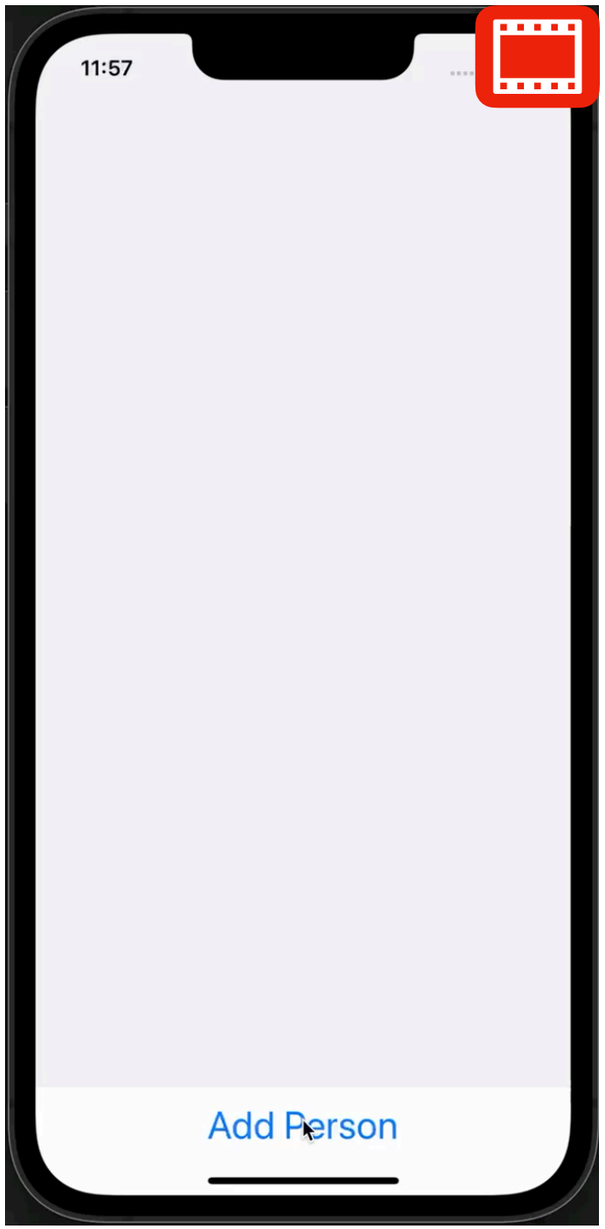
You have probably seen code similar to this before:

```
List(people, id: \.self)
```

You don't have to do this if your items conform to `Identifiable` though.

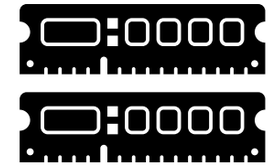
The good news is Core Data automatically makes your entities conform to `Identifiable` for you so you will never have to specify the id.

@Environment for Managed Object Context



We need to reference the managed object context now. We added it to the environment so we will need the **@Environment** property wrapper to access it.

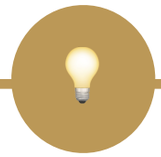
```
struct FirstExampleView: View {
    @FetchRequest(sortDescriptors: []) private var people: FetchedResults<PersonEntity>
    @Environment(\.managedObjectContext) var moc

    var body: some View {
        VStack {
            List(people) { person in
                Text(person.name ?? "")
            }

            Button("Add Person") {
                let person = PersonEntity(context: moc)
                person.name = ["Mark", "Lem", "Chase"].randomElement()
                try? moc.save()
            }
        }
        .font(.title)
    }
}
```



You now have access to the managed object context of your persistent store.

This is your place to add new data, make data changes, delete data, and then decide if you want to save it or undo it.

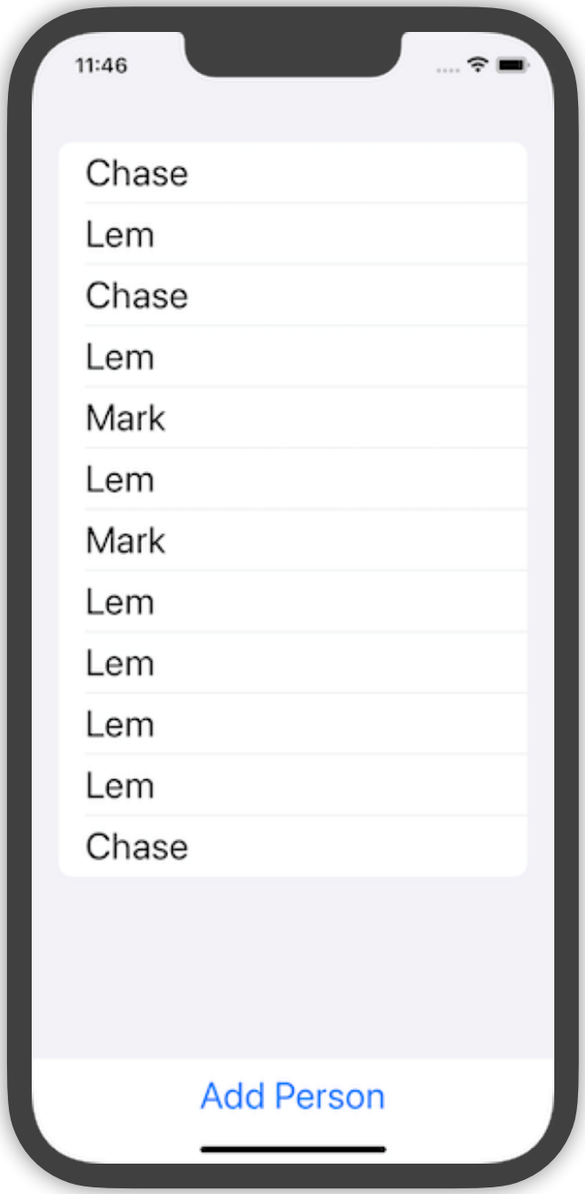
Note: I named the property “moc” for **m**anaged **o**bject **c**ontext. Some also use “context”. It’s up to you.



The save function returns a bool that indicates success or failure. The question mark after `try` means the bool could be nil or have a value.

When a new data object (PersonEntity) is created, you pass in the managed object context it is associated with. That way, when you call the save function on the managed object context, it knows which persistent store to write it to.

Questions & Answers



The @FetchRequest didn't have to somehow connect to the managed object context that we added to the environment?
The @FetchRequest is so smart that it automatically looks for the managed object context in the **environment**.

If it doesn't find one, you will see a purple memory warning in Xcode that says:
"*Context in environment is not connected to a persistent store coordinator*"

This is when you know you forgot to set the managed object context.

How did the view update automatically when new people were created?
All we did was call `save()` on the managed object context. So how did the view update? The @FetchRequest maintains a connection with the context. If data in the context changes, it sends it to the @FetchRequest and the view is updated. Pretty cool, right?

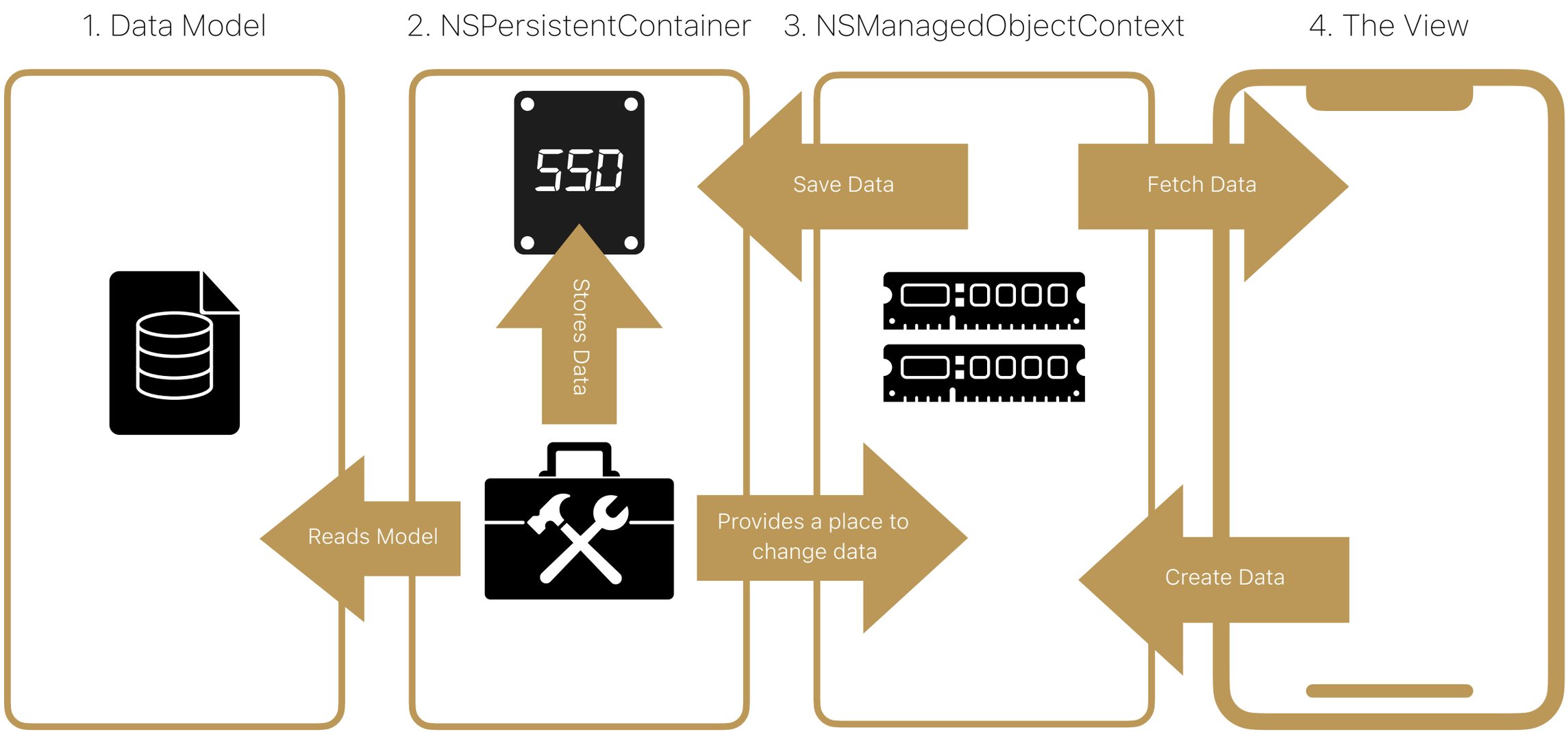
What happens to the data after I quit the app?
When the new data objects were created we called the managed object context's `save()` function. This persisted the object to the store. So if you quit the app **the data is preserved**. If you re-open the app, the data is loaded from the persisted store again and displayed on the screen.

How do I delete this data?
This book will show you how to do that but for now, you can delete your app from the simulator. That will remove the data from the persisted storage that was connected to the app.

PersonEntity.name is optional. Can we make it not optional?
Unfortunately, no. The way Core Data works is that attributes always come back optional. But you will learn nicer ways of handling this in the Displaying Data chapter rather than nil coalescing (`??`) values everywhere.

4 Steps Done!

You have now successfully added data to your persistent store and have data successfully being fetched and shown on the view.



Summary

When looking at the total amount of code we had to create to get Core Data working, there really wasn't all that much between the 3 files.

1

```
class FirstExampleContainer {
    let persistentContainer: NSPersistentContainer

    init() {
        persistentContainer = NSPersistentContainer(name: "FirstExample")
        persistentContainer.loadPersistentStores { _, _ in }
    }
}
```

First, we created a class to create our persistent container.

2

```
@main
struct SwiftUI_CoreDataApp: App {
    var body: some Scene {
        WindowGroup {
            FirstExampleView()
                .environment(\.managedObjectContext, FirstExampleContainer().persistentContainer.viewContext)
        }
    }
}
```

And then we added the managed object context (viewContext property) to the environment's `managedObjectContext`.

Summary

3

```
struct FirstExampleView: View {
    @FetchRequest(sortDescriptors: []) private var people: FetchedResults<PersonEntity>
    @Environment(\.managedObjectContext) var moc

    var body: some View {
        VStack {
            List(people) { person in
                Text(person.name ?? "")
            }

            Button("Add Person") {
                let person = PersonEntity(context: moc)
                person.name = ["Mark", "Lem", "Chase"].randomElement()
                try? moc.save()
            }
        }
        .font(.title)
    }
}
```

Finally, we have our view that used a `@FetchRequest` to get the data.

We also had to add a reference to the `managedObjectContext` in the environment so we could add data.

The Core Data Concepts and First Example chapters were meant to help you understand the main parts of Core Data and to see how easy it can be to get started. This is your foundation to build on.

MOCK DATA

Many times you want to see a preview of the UI you are building WITH mock data. But how can you add mock data to Core Data without it persisting? What if you want to test deleting data? You don't want to keep having to add the same data over and over again.

In this chapter, I'll show you how to set up mock data so you can see it in your SwiftUI Preview.

Data Model

For this chapter, we are going to use a simple data model with one entity and two fields. It is in a file called **FriendsDataModel.xcdatamodeld**.

FriendsDataModel

As a convention from this point on I will be adding "**DataModel**" to all of my data model file names.

I also add "**Entity**" to all entity names so when used elsewhere, it is easy to tell it comes from the data model.

And here is the persistent container class we will start with. This is just like your first example:

```
class FriendsContainer {
    let persistentContainer: NSPersistentContainer

    init() {
        persistentContainer = NSPersistentContainer(name: "FriendsDataModel")
        persistentContainer.loadPersistentStores { _, _ in }
    }
}
```

Remember, the name has to match the **file name** of the data model.

Persistent Store Location

Your persistent store for your app has a path, a URL, to its location. Let's see where it is.

Let's add a new line to our FriendsContainer to see this URL.

```
class FriendsContainer {
    let persistentContainer: NSPersistentContainer

    init() {
        persistentContainer = NSPersistentContainer(name: "FriendsDataModel")


        print(persistentContainer.persistentStoreDescriptions.first!.url!.absoluteString)


        persistentContainer.loadPersistentStores { _, _ in }
    }
}
```

The persistent store descriptions are an array of persistent store objects that have properties to allow you to further define certain aspects of a persistent store, such as its URL (location) to where it is stored on the device.

As you can see the persistent store is a sqlite file with the exact same name as the name of our data model "FriendsDataModel".

We're going to change this URL to a location that allows us to temporarily store mock data.

```
Optional("file:///Users/mark/Library/Developer/CoreSimulator/Devices/FDCCB8F9-79E4-499C-8CFD-2E8243B84B08/data/Containers/Data/Application/490E0318-F97F-4BD5-907B-708B2C0D0ADE/Library/Application%20Support/FriendsDataModel.sqlite")
```

Mock Data Location

Apple gives us a special temporary location for our persistent store where we can load mock data, change, delete, etc. The data only last until the preview redraws or your app stops running in the Simulator.

You want to be able to toggle this temporary path for mock data because your real app will not use this path.

```
class FriendsContainer {
    let persistentContainer: NSPersistentContainer

    init(forPreview: Bool = false) {
        persistentContainer = NSPersistentContainer(name: "FriendsDataModel")

        if forPreview {
            persistentContainer.persistentStoreDescriptions.first!.url = URL(fileURLWithPath: "/dev/null")
        }

        persistentContainer.loadPersistentStores { _, _ in }
    }
}
```

Here is how you set the container's location to a temporary location where you can insert, change and delete data and none of it will persist.

This means every time you run the app with forPreview set to true, it will always start over.

What about inserting mock data though?

Let's do this next...

Mock Data

```
class FriendsContainer {
    let persistentContainer: NSPersistentContainer

    init(forPreview: Bool = false) {
        persistentContainer = NSPersistentContainer(name: "FriendsDataModel")

        if forPreview {
            persistentContainer.persistentStoreDescriptions.first!.url = URL(fileURLWithPath: "/dev/null")
        }

        persistentContainer.loadPersistentStores { _, _ in }

        if forPreview {
            addMockData(moc: persistentContainer.viewContext)
        }
    }
}
```

If forPreview is true then we want to load up some mock data

```
extension FriendsContainer {
    func addMockData(moc: NSManagedObjectContext) {
        let friend1 = FriendEntity(context: moc)
        friend1.firstName = "Chris"
        friend1.lastName = "Bloom"

        let friend2 = FriendEntity(context: moc)
        friend2.firstName = "Jaqueline"
        friend2.lastName = "Cruz"

        let friend3 = FriendEntity(context: moc)
        friend3.firstName = "Rodrigo"
        friend3.lastName = "Jones"

            // ...

        try? moc.save()
    }
}
```

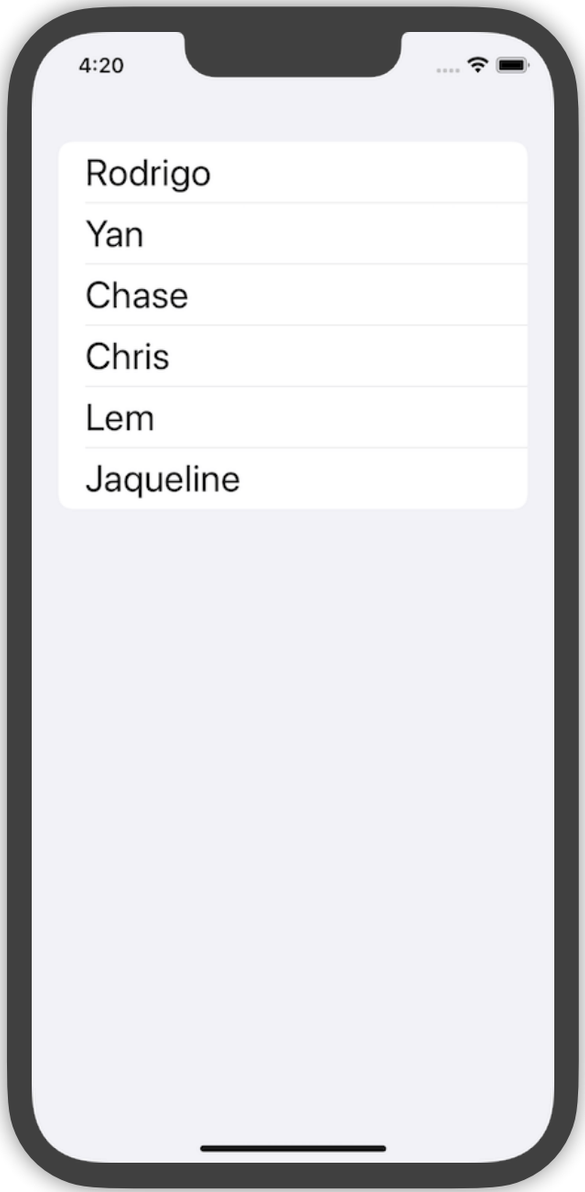
The newly created managed object context is passed in because you need this whenever you create a new entity, such as the `FriendEntity`.

Newly created entities are first added to context (memory) and then are saved and persisted on the disk.

OK, you have a temporary location to play in and some mock data loaded up.

Now let's set up the SwiftUI preview!

SwiftUI Previewing Mock Data




```
struct PreviewingData_Intro: View {
    @FetchRequest(sortDescriptors: []) var friends: FetchedResults<FriendEntity>

    var body: some View {
        List(friends) { friend in
            Text(friend.firstName ?? "")
        }
        .font(.title)
    }
}

struct PreviewingData_Intro_Previews: PreviewProvider {
    static var previews: some View {
        PreviewingData_Intro()
            .environment(\.managedObjectContext,
                         FriendsContainer(forPreview: true).persistentContainer.viewContext)
    }
}
```

This is all you have to add!

Notice we're not adding any data here.

All you have to do is set forPreview to true and data automatically gets added. Nothing gets persisted.

This line is kind of long. **Could it be shorter?**

Static Preview Property

```
class FriendsContainer {
    static var preview: NSManagedObjectContext {
        get {
            let persistentContainer = NSPersistentContainer(name: "FriendsDataModel")
            persistentContainer.persistentStoreDescriptions.first!.url = URL(fileURLWithPath: "/dev/null")
            persistentContainer.loadPersistentStores { _, _ in }
            addMockData(moc: persistentContainer.viewContext)
            return persistentContainer.viewContext
        }
    }

    let persistentContainer: NSPersistentContainer

    init(forPreview: Bool = false) {

        persistentContainer = NSPersistentContainer(name: "FriendsDataModel")

        if forPreview {
            persistentContainer.persistentStoreDescriptions.first!.url = URL(fileURLWithPath: "/dev/null")
        }

        persistentContainer.loadPersistentStores { _, _ in }

        if forPreview {
            FriendsContainer.addMockData(moc: persistentContainer.viewContext)
        }
    }
}
```

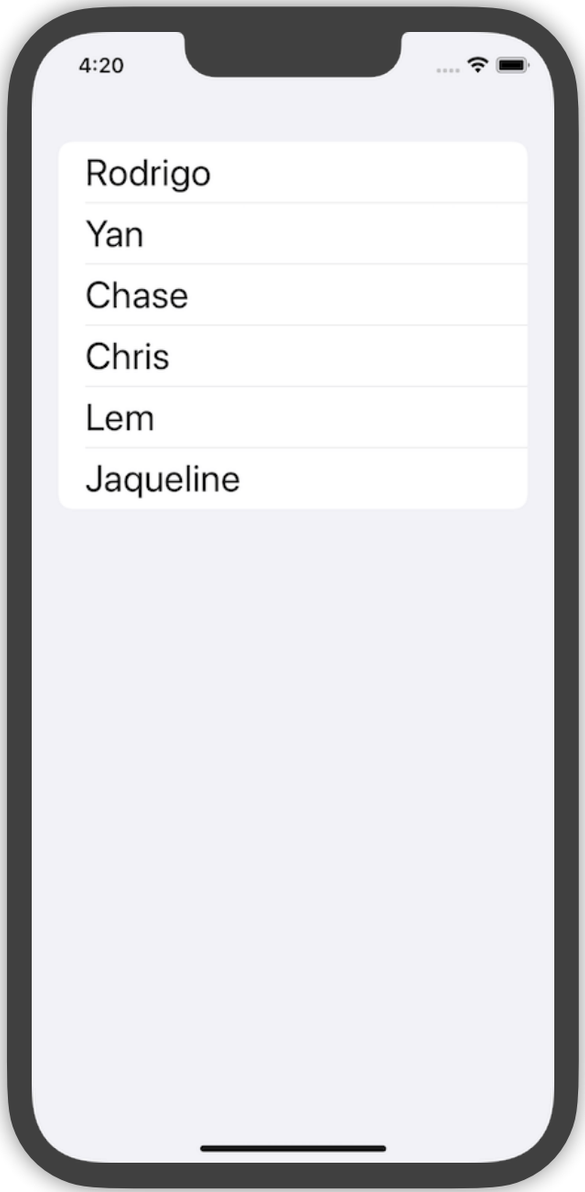
You could create a static property to instantiate your persistent store, load up some mock data and then return your managed object context like this.

Because this preview property is **static**, you will have to make the `addMockData` function static too.

Make the `addMockData` function static and refer to it through the class now.

There are probably thousands of ways you can create your container for your production app and for previewing with mock data.
I wanted to give you some ideas and options here so you could try different things and see what works for you!

Shorter Property for Preview




```
struct PreviewingData_Intro: View {
    @FetchRequest(sortDescriptors: []) var friends: FetchedResults<FriendEntity>

    var body: some View {
        List(friends) { friend in
            Text(friend.firstName ?? "")
        }
        .font(.title)
    }
}

struct PreviewingData_Intro_Previews: PreviewProvider {
    static var previews: some View {
        PreviewingData_Intro()
            .environment(\.managedObjectContext, FriendsContainer.preview)
    }
}
```

Then in your preview, you simply refer to this static property.

DISPLAYING DATA

In your data model, most of your entity attributes will be optional types. You don’t have a choice in this matter.

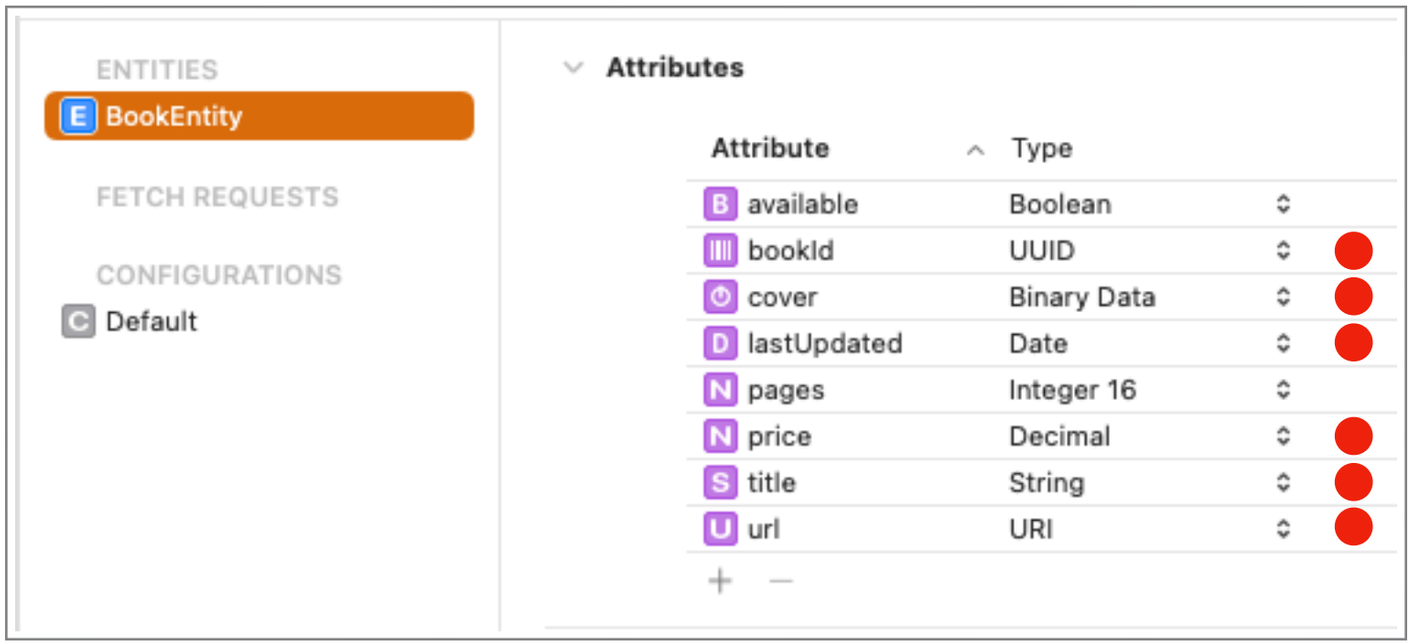
This means when you want to display data, there’s a lot of optional handling because if a value is nil, you have to provide a default value.

You are going to learn how to make this easier.

Data Model

For this chapter we are going to use this data model. It is in a file called **BooksDataModel.xcdatamodeld**.

BooksDataModel



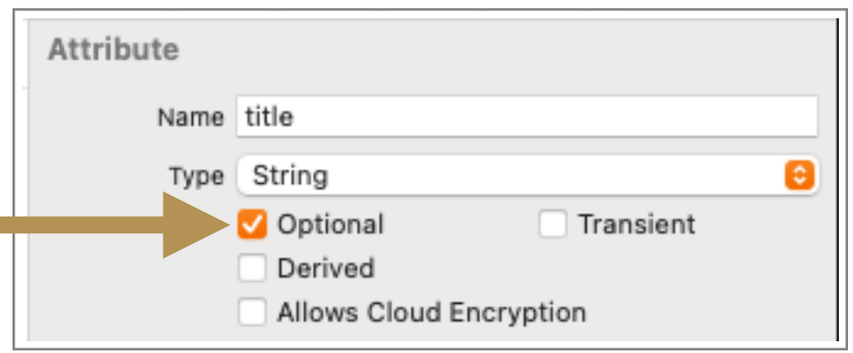
I wanted to have a data model that used a lot of different data types.

Core Data will make a lot of these types optional.

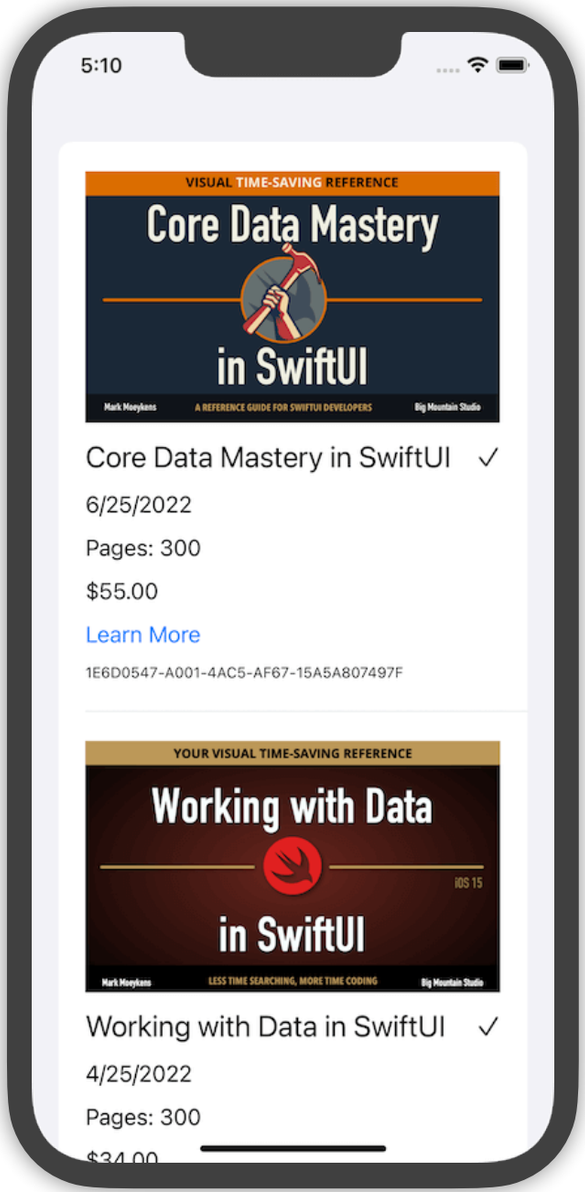
The ones with the red dot will become optional in Swift.

Can I just uncheck Optional in the Data Model Inspector?

No. This does not control whether the attribute type gets converted into a Swift optional type or not.
It determines if an error is thrown when inserting or updating if that value is missing.



Handling Nils




```
struct DisplayData_Intro: View {
   @FetchRequest(sortDescriptors: []) private var books: FetchedResults<BookEntity>

   var body: some View {
      List(books) { book in
         VStack(alignment: .leading, spacing: 12) {
            getImage(imageData: book.cover)
               .resizable()
               .scaledToFit()
            HStack {
               Text(book.title ?? "")
                  .font(.title2)
               Spacer()
               Image(systemName: book.available ? "checkmark" : "xmark")
            }
            Text(book.lastUpdated?.formatted(date: .numeric, time: .omitted) ?? "N/A")
            Text("Pages: \(book.pages )")
            Text((book.price ?? 0) as Decimal, format: .currency(code: "USD"))
            Link(destination: book.url ?? URL(string: "https://www.bigmountainstudio.com")!) {
               Text("Learn More")
            }
            Text(book.bookId?.uuidString ?? "")
               .font(.caption2)
         }
         .padding(.vertical)
      }
   }

   func getImage(imageData: Data?) -> Image {
      if let data = imageData, let image = UIImage(data: data) {
         return Image(uiImage: image)
      } else {
         return Image(systemName: "photo.fill")
      }
   }
}
```

There's a lot of work on the view to handle all of these nils.

Notice Core Data won't make booleans and numbers optional.

There's also a lot of work to not only handle if an image is nil but to convert it into a format SwiftUI can use.

Let's simplify this.

Extending the Entity Class

The goal here is to get the data “display ready”.

Here are some examples of handling nils, images, and formatting.

I prefix all of my properties with “view”.

That way when I use autocomplete, I can just start typing “view” to see all of my properties.

Bookmark this page so that when it comes time for you to format your values you can come here to find examples for most data types.

```
// Handle nils and formatting
extension BookEntity {
    var viewCover: UIImage {
        if let data = cover, let image = UIImage(data: data) {
            return image
        } else {
            return UIImage(systemName: "note.text")! // SF Symbol
        }
    }

    var viewTitle: String {
        title ?? "No Book Title"
    }
    var viewAvailability: String {
        available ? "checkmark" : "xmark"
    }
    var viewLastUpdated: String {
        return "Last Updated: " +
        (lastUpdated?.formatted(date: .numeric, time: .omitted) ?? "N/A") // Stands for "Not Available"
    }
    var viewPages: String {
        "Pages: \(pages)"
    }
    var viewPrice: String {
        let formatter = NumberFormatter()
        formatter.locale = Locale.current
        formatter.numberStyle = .currency
        return formatter.string(from: price ?? 0)!
    }
    var viewUrl: URL {
        url ?? URL(string: "https://www.bigmountainstudio.com")!
    }
    var viewBookId: String {
        bookId?.uuidString ?? ""
    }
}
```

All the entities defined in the Data Model file get converted into classes.

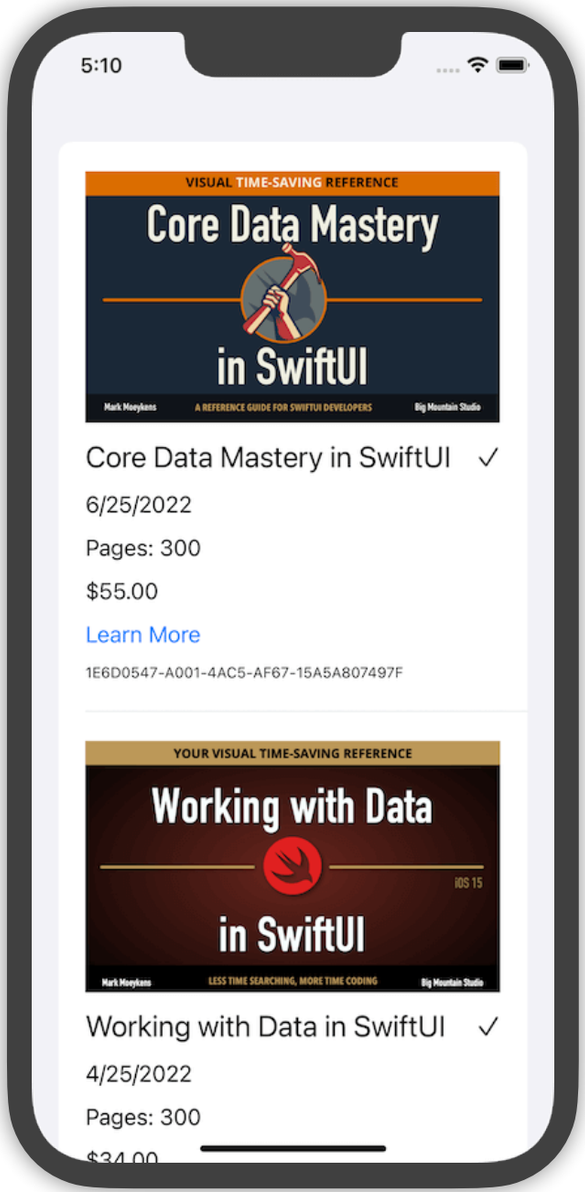
Xcode “synthesizes” or autogenerates the entity and attribute information into Swift classes and properties.

You can then extend or add on to those autogenerated classes to add your own properties and functions.

Note: You never want to edit the autogenerated code or else you will most likely lose your changes if the entity gets regenerated again.

Create an extension instead.

SwiftUI View with Extended Properties



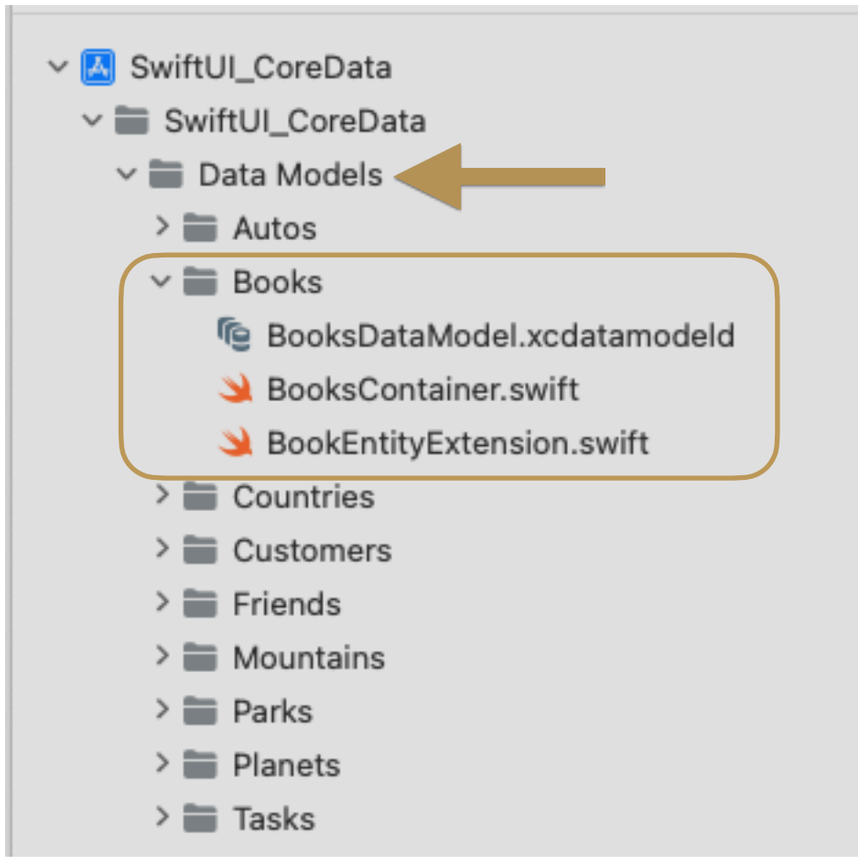

```
struct DisplayData_HandlingNils: View {
    @FetchRequest(sortDescriptors: []) private var books: FetchedResults<BookEntity>

    var body: some View {
        List(books) { book in
            VStack(alignment: .leading, spacing: 12) {
                Image(uiImage: book.viewCover)
                    .resizable()
                    .scaledToFit()
                HStack {
                    Text(book.viewTitle)
                        .font(.title2)
                    Spacer()
                    Image(systemName: book.viewAvailability)
                }
                Text(book.viewLastUpdated)
                Text(book.viewPages)
                Text(book.viewPrice)
                Link(destination: book.viewUrl) {
                    Text("Learn More")
                }
                Text(book.viewBookId)
                    .font(.caption2)
            }
            .padding(.vertical)
        }
    }
}
```

Look how much cleaner this all looks compared to the version we started with.

All of the formatting logic has been extracted into the entity extension class.

Process & File Organization

File Organization



Xcode Project Navigator

I keep my data model, container, and entity extensions all in the same folder.

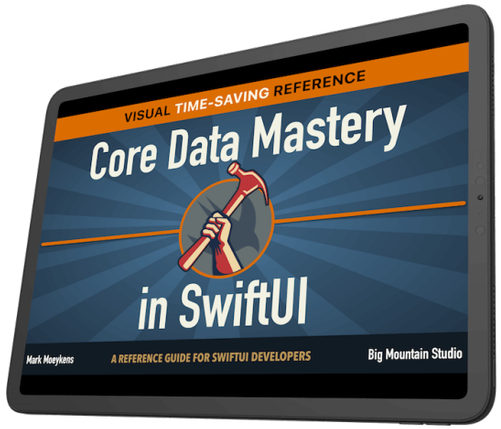
If you are using the companion Xcode project, you will find all the data models and related files in the **Data Models folder** and within their own folders from there.

Remember:

- **Data Model** - Tells Core Data what the data will look like.
- **Container** - Creates your persistent container and gives you a managed object context.
- **Entity Extensions** - Extends the autogenerated entity classes with your own properties to format the values and handle nils for use on your UI.

In your own app, you will most likely only have **ONE** data model, container, and as many entity extensions as entities in your data model.

WHERE TO GO FROM HERE



Thank you for reading the Core Data Quick Start in SwiftUI book.

This is just the BEGINNING of a larger book called **Core Data Mastery in SwiftUI**. In the Mastery book you will learn much more about Core Data and be able to answer questions such as:

- How do I create fetch requests that can filter and sort my data?
- Can I fetch data that can be bound to a List with sections?
- Is there a way to add data validation to my data model and show the results in my SwiftUI views?
- How do I use the managed object context to insert, update, delete or check for changes before saving?
- How do you use concurrency in Core Data?
- What can I do to allow users to undo and redo Core Data changes?
- Can I split out my data model into multiple entities and connect them with relationships?
- How do I use Core Data with observable objects and SwiftUI views?
- Can I version my data model with changes for new versions of my app?
- How can I use Core Data and sync with iCloud between devices?

THE END